SC33-0006-7
File No. S370-29

Program Product

OS PL/I Optimizing Compiler: Programmer's Guide

Optimizing Compiler	5734-PL1
Resident Library	5734-LM4
Transient Library	5734-LM5

(These program products are also available as composite package 5734-PL3)

IBM

SC33-0006-7
File No. S370-29

Program Product

OS PL/I Optimizing Compiler: Programmer's Guide

Optimizing Compiler	5734-PL1
Resident Library	5734-LM4
Transient Library	5734-LM5

(These program products are also available as composite package 5734-PL3)

Release 4.0
Release 5.0
Release 5.1

Seventh Edition (September 1985)

This is a major revision of, and makes obsolete, SC33-0006-6.

This edition applies to Releases 4.0, 5.0, and 5.1 of the OS PL/I Optimizing Compiler, Program Product 5734-PL1, the OS PL/I Resident Library, Program Product 5734-LM4, the OS PL/I Transient Library, Program Product 5734-LM5, and composition package, Program Product 5734-PL3, and to any subsequent releases until otherwise indicated in new editions or technical newsletters.

The changes for this edition are summarized under "Summary of Amendments" following the preface. Specific changes are indicated by a vertical bar to the left of the change. These bars will be deleted at any subsequent republication of the page affected. Editorial changes that have no technical significance are not noted.

Changes are made periodically to this publication; before using this publication in connection with the operation of IBM systems, consult the latest IBM System/370 and 4300 Processors Bibliography, GC20-0001, for the editions that are applicable and current.

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 program product in this publication is not intended to state or imply that only IBM's program product may be used. Any functionally equivalent program may be used instead.

Publications are not stocked at the address given below; requests for IBM publications should be made to your IBM representative or to the IBM branch office serving your locality.

A form for readers' comments is provided at the back of this publication. If the form has been removed, comments may be addressed to IBM Corporation, P.O. Box 50020, Programming Publishing, San Jose, California, U.S.A. 95150. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

PREFACE

This publication explains how to use the compiler to execute PL/I programs and describes the operating system features that can be required by a PL/I programmer. It is a guide to the use of the OS PL/I Optimizing Compiler (Program No. 5734-PL1) in a batch environment of your operating system. It does not describe the language implemented by the compiler, nor does it explain how to use the compiler in an operating system with the Time Sharing Option (TSO), or with the Conversational Monitor System (CMS) of VM/370; these are the functions of the manuals listed under "Associated Publications" on page iv.

For execution of a PL/I program, the optimizing compiler employs subroutines from the OS PL/I Resident Library (Program No. 5734-LM4) and the OS PL/I Transient Library (Program No. 5734-LM5), and this programmer's guide assumes the availability of these program products.

Different release levels of the OS PL/I Optimizing Compiler and the PL/I Resident and Transient libraries will be compatible in execution provided that the following conditions are satisfied:

1. The release and service level of the transient library is equal to or greater than the release and service level of the resident library.

2. The release and service level of the resident library is equal to or greater than the release and service level of the compiler.

ORGANIZATION OF THIS BOOK

Chapter 1, "Introduction" explains how to run a PL/I program. The rest of the manual contains more detailed information on the optimizing compiler, and provides guidance and reference information on operating system features that are likely to be required by the PL/I applications programmer. Most of this information is equally relevant to the use of the compiler in a batch or conversational (TSO or CMS) environment.

Chapter 2, "The Compiler" describes the optimizing compiler, the data sets it requires, its optional facilities, and the listings it produces.

Chapter 3, "The Linkage Editor and the Loader" contains information for the linkage editor and loader that is similar to Chapter 2, "The Compiler." Either the linkage editor or the loader is needed in addition to the compiler to prepare a PL/I program for execution.

Chapter 4, "Data Sets and Files" through Chapter 7, "Using VSAM Data Sets from PL/I" on page 222 are concerned with the various types of data sets that can be created and accessed by a PL/I program, and explains how to define these data sets.

Chapter 8, "Libraries of Data Sets" describes libraries of data sets.

Chapter 9, "Cataloged Procedures" describes the cataloged procedures provided by IBM for the optimizing compiler, and explains how to modify them.

Chapter 10, "Program Checkout" deals with the facilities available for debugging PL/I programs.

Chapter 11, "Communicating between PL/I and Assembler-Language Modules" and Chapter 14, "Interlanguage Communication with COBOL and FORTRAN" describe the language implemented by the optimizing

compiler to facilitate communication between programs written in PL/I and those written in FORTRAN, COBOL, and Assembler language.

Chapter 12, "The Sort Program" and
Chapter 13, "Checkpoint/Restart" are concerned with the use of built-in subroutines included in the resident library to provide direct interface between PL/I programs and the operating system sort/merge and checkpoint/restart facilities.

Chapter 15, "Using PL/I on CICS" tells how to use PL/I under CICS. The user who is running a PL/I application under CICS must read Chapter 15, "Using PL/I on CICS." The chapter lists restrictions for running under CICS, and describes the differences from batch operation that exist when running under CICS.

A series of appendixes supplies sundry reference information.

ASSOCIATED PUBLICATIONS

OS PL/I

- OS and DOS PL/I Language Reference Manual, GC26-3977

  Describes the language implemented by the optimizing compiler.

- OS PL/I Optimizing Compiler: General Information, GC33-0001

  Gives an overview of the optimizing compiler.

- OS PL/I Optimizing Compiler: TSO User's Guide, SC33-0029

  Describes how to use the optimizing compiler in a TSO environment.

- OS PL/I Optimizing Compiler: CMS User's Guide, SC33-0047

  Describes how to use the optimizing compiler in a CMS environment.

- OS PL/I Optimizing Compiler: Messages, SC33-0027

  Contains the diagnostic messages issued by the compiler and the transient library. It also contains any necessary explanation of the message with the suggested programmer response.

- OS and DOS PL/I Optimizing Compilers: Debug Guide, SY26-3990

  Aids in problem determination.

- OS PL/I Optimizing Compiler: Execution Logic, SC33-0025

  Describes how a compiled program is executed.

- OS PL/I Optimizing Compiler: Installation Guide, SC33-0026. (For OS PL/I Release 4)

- OS PL/I Optimizing Compiler: Installation Guide for MVS, SC26-4121.

- OS PL/I Optimizing Compiler: Installation Guide for CMS, SC26-4122. (For OS PL/I Release 5.1)

  Explains how to install the compiler.

- OS PL/I Checkout Compiler: Programmer's Guide, SC33-0007

Contains information about the OS PL/I Checkout Compiler and about combining modules from the optimizing and checkout compilers.

CICS

- Customer Information Control System/ Virtual Storage (CICS/VS) Version 1 Release 6: Application Programmer's Reference Manual (Macro Level), SC33-0079. (For OS PL/I Release 4 only)
- Customer Information Control System/ Virtual Storage (CICS/VS) Version 1 Release 6: Application Programmer's Reference Manual (Command Level), SC33-0077. (For OS PL/I Release 4 only)
- CICS/OS/VS Version 1 Release 6 Modification 1 Application Programmer's Reference Manual (Command Level), SC33-0161.

COBOL

- OS/VS COBOL Compiler and Library Programmer's Guide, SC28-6483.
- VS COBOL II Application Programming Guide, GC26-4045.

VS FORTRAN

- VS FORTRAN Application Programming: Guide, SC26-3985.
- VS FORTRAN Application Programming: Library Reference, SC26-3989.

IMS/VS

- IMS/VS Version 1 Application Programming, SH20-9026.
- IMS/VS Version 1 Data Base Administration Guide, SH20-9025.

IBM DATABASE2

- IBM DATABASE2 Application Programming Guide for CICS/OS/VS Users, SC26-4080
- IBM DATABASE2 Application Programming Guide for IMS/VS Users,SC26-4079
- IBM DATABASE2 Application Programming Guide for TSO Users, SC26-4081
- IBM DATABASE2 Introduction to SQL, GC26-4082

MVS

MVS/System Product V1.2.1

- OS/VS Linkage Editor and Loader, GC26-3813.

MVS/Extended Architecture

- MVS/Extended Architecture Conversion Notebook, GC28-1143.
- MVS/Extended Architecture Linkage Editor and Loader, GC26-4011 .
- MVS/Extended Architecture System Programming Library: System Modifications,CG28-1152.
- MVS/Extended Architecture System Programming Library: User Exits, GC28-1147.
- MVS/Extended Architecture System Programming Library: 31-Bit Addressing, GC28-1158.

DFSORT

- DFSORT Application Programming: Guide, SC33-4035.
- Getting Started with DFSORT, SC26-4109.

OS/VS Sort/Merge (Sort/Merge Release 5 only)

- OS/VS Sort/Merge Programmer's Guide, SC33-4035.

VSAM

- MVS/Extended Architecture VSAM Administration Guide, GC26-4015
- OS/VS Virtual Storage Access Method (VSAM) Programmer's Guide, GC26-3838.

RELATED RECOMMENDED PUBLICATIONS

A number of system publications are referred to throughout the manual by generic names such as "the linkage editor manuals." The actual manual you require will depend on your installation.

When using this manual, you should have the job control language (JCL) reference manual for your operating system, and the linkage editor manual.

For information on the 3800 Printing Subsystem, see the IBM 3800 Printing Subsystem Programmer's Guide, GC26-3846.

For definitions of terms used in this manual, see the IBM Vocabulary for Data Processing, Telecommunications, and Office Systems, GC20-1699.

SYNTAX NOTATION

Throughout this publication, whenever a PL/I statement—or some other combination of elements—is discussed, the manner of writing that statement or phrase is illustrated with a uniform system of notation. This notation is **not** a part of PL/I; it is merely a notation that is used to describe the syntax, or construction, of the language.

For the syntax notation used in this publication, see the "Syntax Notation" section of the OS and DOS PL/I Language Reference Manual.

INDUSTRY STANDARDS

The OS PL/I Optimizing Compiler is designed according to the specifications of the following industry standards as understood and interpreted by IBM as of May, 1979:

- American National Standard Code for Information Interchange (ASCII), X3.4 - 1977.
- American National Standard Representation of Pocket Select Characters in Information Interchange, level 1, X3.77 - 1980 (proposed to ISO, March 1, 1979)
- The draft proposed American National Standard Representation of Vertical Carriage Positioning Characters in Information Interchange, level 1, dpANS X3.78 (Also proposed to ISO, March 1, 1979)

SUMMARY OF AMENDMENTS

SEPTEMBER 1985

NEW PROGRAMMING SUPPORT

Information on using the 31-bit addressing capability of MVS/XA for PL/I Release 5.1 running under CICS/OS/VS Version 1, Release 6, Modification 1, with upgrade, has been added in Chapter 15, "Using PL/I on CICS" on page 360. That chapter has been rewritten to include information that formerly appeared in a separate appendix.

Support of OS PL/I Release 5.1 for VM/SP and VM/PC is also provided.

SERVICE CHANGES

Miscellaneous corrections have been made throughout the manual.

OCTOBER 1984

EXTENDED ARCHITECTURE SUPPORT

Information on running the optimizing compiler, its generated object code, and libraries in an MVS Extended Architecture (MVS/XA) environment has been added in a new Appendix.

The new ISAINC, HEAP, and TASKHEAP execution-time options have been added to the "The Compiler" chapter.

Enhanced error-handling support for OS PL/I transactions under IMS/VS Releases 1.2 and 1.3, and support for the 31-bit addressing capabilities of MVS/XA, with IMS/VS Release 1.3, are described in another new Appendix.

> **Note:** OS PL/I Optimizing Compiler and Libraries Release 5.0 will not support VM/CMS and CICS/VS. Users of these products will continue to be supported on OS PL/I Release 4.0.
>
> Release 4 of the PL/I Optimizing Compiler and Libraries is the last release to support VS1.
>
> For Release 5.0, the storage size of the compiler is increased to 128K.

SERVICE CHANGES

Figures have been renumbered to enhance retrievability of information. Page numbers have also been added to the heading and figure references to improve clarity and retrievability.

SEPTEMBER 1981

NEW PROGRAMMING SUPPORT

For Extended Graphic Character Set support, the GRAPHIC compiler option and the GRAPHIC ENVIRONMENT option are described.

SERVICE CHANGES

This edition is for use with the new OS and DOS PL/I Language Reference Manual, order number GC26-3977.

Information moved from the old OS PL/I Checkout and Optimizing Compilers: Language Reference Manual into this edition includes:

- "The ENVIRONMENT Attribute," data transmission statements, and related topics.
- Chapter 4, "Data Sets and Files" on page 100
- Chapter 14, "Interlanguage Communication with COBOL and FORTRAN" on page 343

Chapter 15, "Using PL/I on CICS" on page 360 has been added. It updates and adds to information formerly in Appendix H. Appendix H now contains "PL/I-CICS System Information."

Appendix A, "VSAM Background" on page 383 has been added. It contains information formerly in Chapter 9.

The appendix on "Running Under a Virtual Storage Operating System (OS/VS)" has been deleted.

Other miscellaneous corrections have been made throughout the publication.

JULY 1979

SERVICE CHANGES

For Release 3, Modification 1, the storage size in which the compiler runs has been increased.

Parts of Chapter 1 that were outdated have been deleted, and the former Chapter 2 has been merged into Chapter 1. Chapter 3 has been deleted, because the information on how to create and access a data set is elsewhere (such as in the job control language manual for your system).

Appendix A, which describes DCB subparameters, has been deleted. Your job control language reference manual contains more up-to-date information on the DCB subparameters.

Appendix B, which described compatibility with the PL/I (F) compiler, has been deleted; this information is in OS PL/I Optimizing Compiler: General Information.

Other miscellaneous corrections have been made throughout the manual.

JUNE 1978

SERVICE CHANGES

A new section has been added to Chapter 5 on link-editing multiple object modules. Various maintenance corrections have been made.

CONTENTS

FIGURES

CHAPTER 1. INTRODUCTION

The process of executing a PL/I program requires a minimum of two job steps.

A compilation job step is always required. In this step the optimizing compiler translates the PL/I source program into a set of machine instructions called an object module. This object module does not include all the machine instructions required to represent the source program. In many instances the compiler merely inserts references to subroutines that are stored in the OS PL/I Resident Library.

To include the required subroutines from the resident library, the object module must be processed by one of two processing programs, the linkage editor or the loader.

When using the linkage editor, two further job steps are required after compilation. In the first of these steps, the linkage editor converts the object module into a form suitable for execution, and includes subroutines, referred to by the compiler, from the resident library. The program in this form is called a load module. In the final job step, this load module is loaded into main storage and executed.

When using the loader, only one more job step is required after compilation. The loader processes the object module, includes the appropriate resident library subroutines, and executes the resultant executable program immediately.

Both the linkage editor and the loader can combine separately produced object modules and previously processed load modules. However, they differ in one important respect: the linkage editor produces a load module, which it always places in a library, where it can be permanently stored and called whenever it is required; the loader creates only temporary executable programs in main storage, where they are executed immediately.

The linkage editor also has several facilities that are not provided by the loader; for example, it can divide a program that is too large for the space available in main storage, so that it can be loaded and executed segment by segment.

The loader is intended primarily for use when testing programs and for processing programs that will be executed only once.

Subroutines from the resident library may contain references to other subroutines stored in the OS PL/I Transient Library. The subroutines from the transient library do not become a permanent part of a load module; they are loaded into main storage when needed during execution of the PL/I program, and the storage they occupy is released when they are no longer needed.

The job control statements shown in Figure 1 on page 2 are sufficient to compile and execute a PL/I program that comprises only one external procedure.

This program uses only punched-card input and printed output. The listing produced includes only the default items. Many other items can be included by specifying the appropriate compiler options in the EXEC statement. The compiler listing and all the compiler options are described in Chapter 2, "The Compiler" on page 3. The linkage editor listing and the linkage editor options are described in Chapter 3, "The Linkage Editor and the Loader" on page 65. Appendix D, "Sample Program" on page 407 is a sample PL/I program that includes most of the listing items discussed in these two chapters.

The example in Figure 1 uses the cataloged procedure PLIXCLG. Several other cataloged procedures are supplied by IBM for use with the optimizing compiler (for example, for compilation only). The use of these other cataloged procedures is described in Chapter 9, "Cataloged Procedures" on page 273.

An alternative method of specifying compiler options is by use of the PROCESS statement, which is described in "Specifying Compiler Options in the *PROCESS Statement" on page 13. An example of a PROCESS statement is:

```
* PROCESS MACRO, OPT(TIME);
```

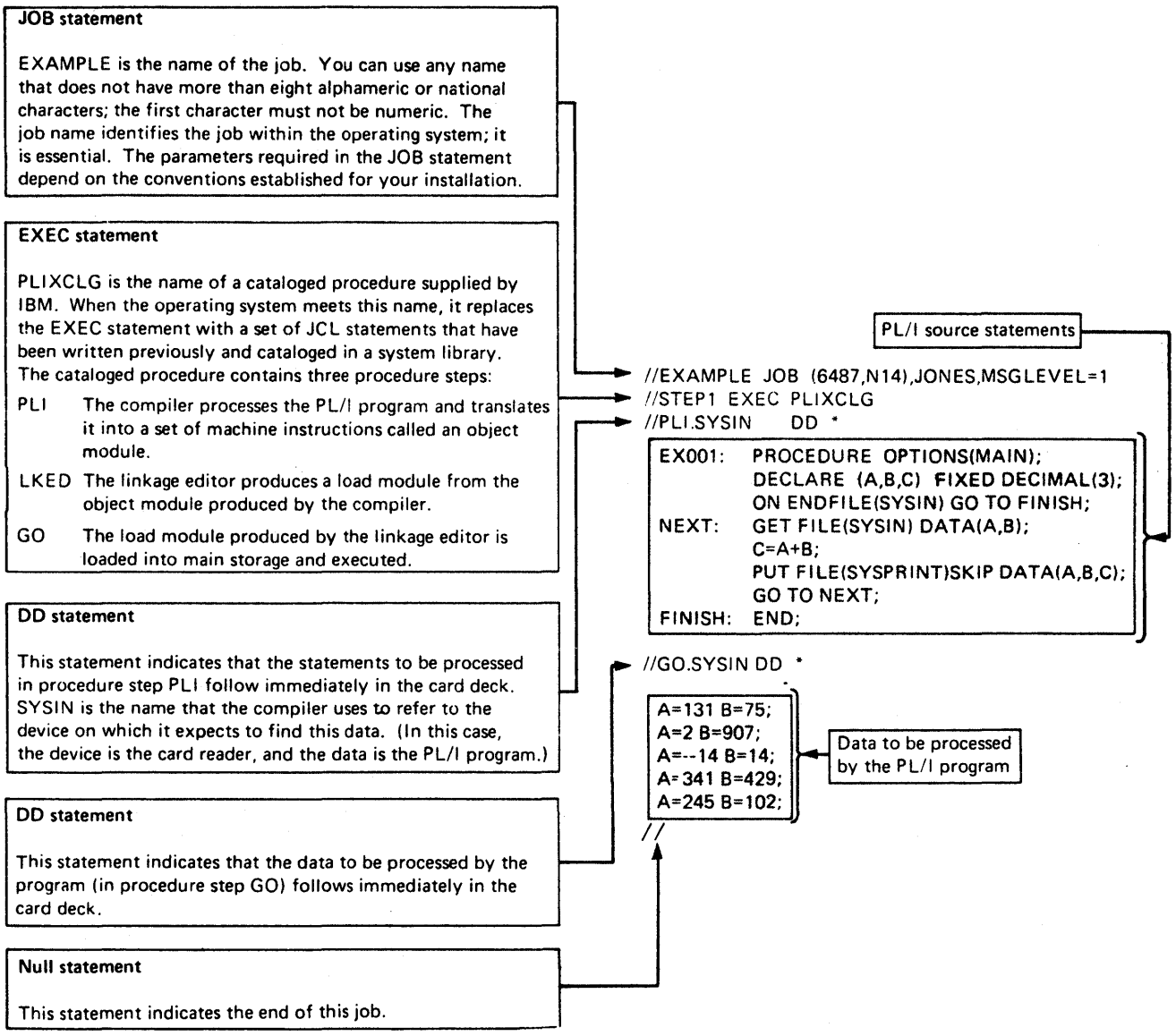


Figure 1. Example of Running a PL/I Program

CHAPTER 2. THE COMPILER

This chapter describes the optimizing compiler and the job control statements required to invoke it, and defines the data sets it uses. It describes the compiler options, the listing produced by the compiler, batched compilation, and the preprocessor, all of which are introduced briefly below.

The optimizing compiler translates the PL/I statements of the source program into machine instructions. A set of machine instructions such as is produced for an external PL/I procedure by the compiler is termed an object module. If several sets of PL/I statements, each set corresponding to an external procedure and separated by appropriate control statements, are present, the compiler can create two or more object modules in a single job step.

However, the compiler does not generate all the machine instructions required to represent the source program. Instead, for frequently used sets of instructions such as those that allocate main storage or those that transmit data between main storage and auxiliary storage, it inserts into the object module references to standard subroutines. These subroutines are stored either in the OS PL/I Resident Library or in the OS PL/I Transient Library.

An object module produced by the compiler is not ready for execution until the appropriate subroutines from the resident library have been included; this is the task of either one of two processing programs, the linkage editor or the loader, described in Chapter 3, "The Linkage Editor and the Loader" on page 65. An object module that has been processed by the linkage editor is referred to as a load module; an object module that has been processed by the loader is referred to as an executable program.

Subroutines from the transient library do not form a permanent part of the load module or executable program. Instead, they are loaded as required during execution, and the storage they occupy is released when they are no longer needed.

While it is processing a PL/I program, the compiler produces a listing that contains information about the program and the object module derived from it, together with messages relating to errors or other conditions detected during compilation. Much of this information is optional, and is supplied either by default or by specifying appropriate options when the compiler is invoked.

The compiler also includes a preprocessor (or compile-time processor) that enables you to modify source statements or insert additional source statements before compilation commences.

Compiler options, discussed further in "Compiler Options" on page 11, can be used for purposes other than to specify the information to be listed. For example, the preprocessor can be used independently to process source programs that are to be compiled later, or the compiler can be used merely to check the syntax of the statements of the source program. Also, continuation of processing through syntax checking and compilation can be made conditional on successful preprocessing.

DESCRIPTION OF THE COMPILER

The compiler consists of a number of load modules, referred to as phases, each of which can be loaded individually into main storage for execution. A simplified flow diagram is shown in Figure 2 on page 5. The first phase to be loaded is a resident control phase, which remains in main storage throughout compilation. This phase consists of a number of service routines that provide facilities required during execution of the remaining phases. One of these routines communicates with the supervisor program of the operating system for the sequential loading of the remaining phases, which are referred to as processing phases.

The resident control phase also causes a transient control phase to be loaded, the function of which is to initialize the operating environment in accordance with your options.

Each processing phase performs a single function or a set of related functions. Some of these phases must be loaded and executed for every compilation; the requirement for other phases depends on the content of the source program or on the optional facilities selected. Apart from the phases that provide diagnostic information, each phase is executed once only.

Input to the compiler is known throughout all stages of the compilation process as text. Initially, this text comprises the PL/I statements of the source program. At the end of compilation, it comprises the machine instructions substituted by the compiler for the source text, together with the inserted references to resident library subroutines for use by the linkage editor or by the loader.

The source text must be in the form of a data set defined by a DD statement with the name SYSIN. The source text is passed to the syntax-analysis stage either directly or after processing by one of the following preprocessor phases:

1. If the source text is in the PL/I 48-character set or in BCD, the 48-character-set preprocessor translates it into the 60-character set. To use the 48-character-set preprocessor, specify the CHARSET(48) or CHARSET(BCD) options.

2. If the source text contains preprocessor statements, the preprocessor executes these statements in order to modify the source text or to introduce additional statements. Also, if the source text is in the PL/I 48-character set or in BCD (as specified by the CHARSET(48) or CHARSET(BCD) options), the preprocessor translates it into the 60-character set. To use the preprocessor, specify the MACRO compiler option.

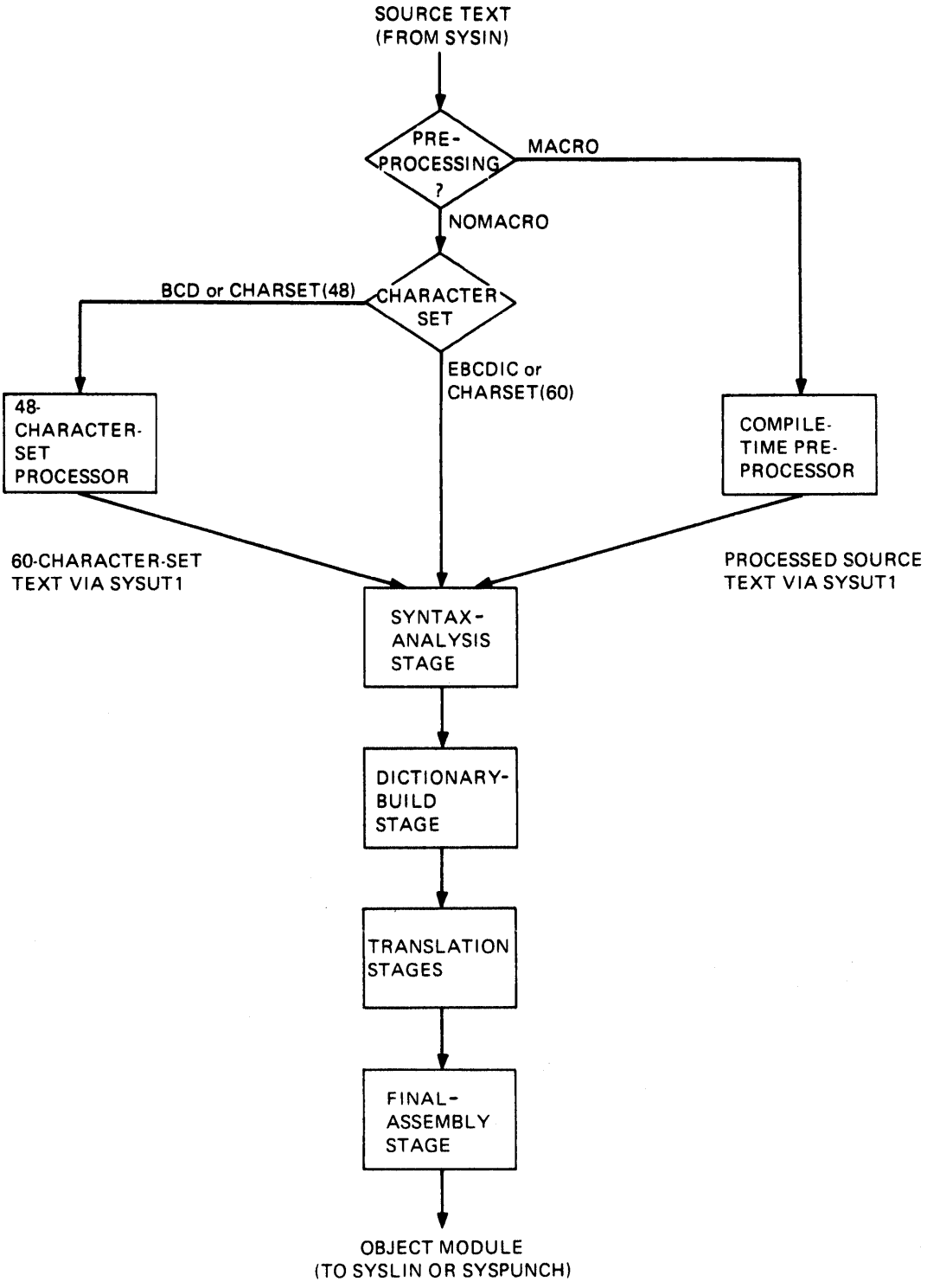


Figure 2. Simplified Flow Diagram of the Compiler

Both preprocessor phases store the translated source text in the data set defined by the DD statement with the name SYSUT1.

The syntax-analysis stage takes its input either from this data set or from the data set defined by the DD statement with the name SYSIN. This stage analyzes the syntax of the PL/I statements and removes any comments and non-significant blank characters.

After syntax analysis, the dictionary-build stage creates a dictionary containing entries for all identifiers in the source text. The compiler uses this dictionary to communicate descriptions of the elements of the source text and the object module between phases. The dictionary-build stage of the compiler replaces all identifiers and attribute declarations in the source text with references to dictionary entries.

Further processing of the text involves several compiler stages, known as translation stages, which:

- Translate the text from the PL/I syntactic form into an internal syntactic form.
- Rearrange the text to facilitate further translation (for example, by replacing array assignments with do-loops that contain element assignments).
- Map arrays and structures to ensure correct boundary alignment.
- Translate the text into a series of fixed-length tables, each with a format that can be used to define machine instructions.
- Allocate main storage for static variables and generate inline code to allow storage to be allocated during execution. (In certain cases resident library subroutines may also be called to allocate storage during execution.)

Standard ddname	Contents of Data Set	Possible Device Classes[1]	Record Format (RECFM)[3]	Record Size (LRECL)[5]	BUFNO Buffers	BLKSIZE Buffers
SYSIN (or SYSCIN)[4]	Input to the compiler	SYSSQ	F,FB,U VB,V	<101(100) <105(104)	2	200
SYSLIN	Object Module	SYSSQ	FB	80	2	80
SYSPUNCH	Preprocessor Output, Compiler Output	SYSSQ SYSCP	FB	80	2	80
SYSUT1[2]	Temporary Workfile	SYSDA	F	1091, 1691, 3491, or 4051 according to available space	-	-
SYSPRINT	Listing, including messages	SYSSQ	VBA	125	2	129

Figure 3 (Part 1 of 2). Compiler Standard Data Sets

Standard ddname	Contents of Data Set	Possible Device Classes[1]	Record Format (RECFM)[3]	Record Size (LRECL)[5]	BUFNO Buffers	BLKSIZE Buffers
SYSLIB[2]	Source statements for preprocessor	SYSDA	F,FB,U V,VB	<101 <105	-	-

Figure 3 (Part 2 of 2). Compiler Standard Data Sets

Notes to Figure 3:

1 The possible device classes are:

SYSSQ Magnetic-tape or direct-access device
SYSDA Direct-access device
SYSCP Card-punch device

2 Any block size can be specified except for SYSLIB and SYSUT1. Block size for SYSLIB depends on the options specified. If the INCLUDE option is specified, the maximum block size is 4260 bytes. If MACRO is specified, the block size maximum is eleven bytes less than the value of LRECL for SYSUT1. The block size for SYSUT1 is always provided by the compiler. The relationship between available space and the LRECL for SYSUT1 is given under "Temporary Workfile (SYSUT1)" on page 9.

3 If the record format is not specified in a DD statement, the default value (underlined) is provided by the compiler.

4 The compiler will attempt to obtain source input from SYSCIN if a DD statement for this data set is provided. Otherwise it will obtain it's input from SYSIN.

5 The numbers in parentheses in the "Record size" column are the defaults which can be overridden by the user.

The final-assembly stage translates the text tables into machine instructions, and creates the external symbol dictionary (ESD) and relocation dictionary (RLD) required by the linkage editor and by the loader.

The external symbol dictionary includes the names of subroutines that are referred to in the object module but are not part of the module and that are to be included by the linkage editor or by the loader; these names, which are termed external references, include the names of all the PL/I resident library subroutines that will be required when the object module is executed. (These resident library subroutines may, in their turn, contain external references to other resident library subroutines required for execution).

The relocation dictionary contains information that enables absolute storage addresses to be assigned to locations within the load module when it is loaded for execution.

The external symbol dictionary and the relocation dictionary are described in Chapter 3, "The Linkage Editor and the Loader" on page 65, which also explains how the linkage editor and the loader use them.

JOB CONTROL STATEMENTS FOR COMPILATION

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

The IBM-supplied cataloged procedures that include a compilation procedure step are as follows:

PLIXC Compile only.

PLIXCL Compile and link-edit.

PLIXCLG Compile, link-edit, and execute.

PLIXCG Compile, load, and execute.

The following paragraphs describe the job control statements needed for compilation. The IBM-supplied cataloged procedures described in Chapter 9, "Cataloged Procedures" on page 273 contain these statements. You will not therefore have to code them yourself unless you are not using the cataloged procedures.

EXEC STATEMENT

The basic EXEC statement is:

```
//stepname EXEC PGM=IEL0AA
```

The PARM parameter of the EXEC statement can be used to specify one or more of the optional facilities provided by the compiler. These facilities are described under "Specifying Compiler Options in the EXEC Statement" on page 12.

DD STATEMENTS FOR THE STANDARD DATA SETS

The compiler requires several standard data sets, the number depending on the optional facilities specified. You must define these data sets in DD statements with the standard ddnames which are shown, together with other characteristics of the data sets, in Figure 3 on page 6. The DD statements SYSIN, SYSUT1, and SYSPRINT are always required.

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

Input (SYSIN, OR SYSCIN)

Input to the compiler must be a data set defined by a DD statement with the name SYSIN or SYSCIN; this data set must have CONSECUTIVE organization. The input must be one or more external PL/I procedures; if you want to compile more than one external procedure in a single job or job step, precede each procedure, except possibly the first, with a PROCESS statement. For further detail, see "Batched Compilation" on page 55.

Eighty-column punched cards are commonly used as the input medium for PL/I source programs. However, the input data set may be on a direct-access device, magnetic tape, or paper tape. The input data set may contain either fixed-length records, blocked or unblocked, variable-length records, or undefined-length records; the maximum record size is 100 bytes. The compiler always reserves 200 bytes of main storage (100 bytes each) for two buffers for this data set; however, you may specify a block size of more than 100 bytes, provided that

sufficient main storage is available to the compiler. (For further details of the SIZE compiler option under "SIZE Option" on page 26.)

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

Output (SYSLIN, SYSPUNCH)

Output (that is, one or more object modules) from the compiler can be stored in either the data set defined by the DD statement with the name SYSLIN (if you specify the OBJECT compiler option) or in the data set defined by the DD statement with the name SYSPUNCH (if you specify the DECK compiler option). You may specify both options in one program, when the output will be stored in both data sets.

The object module is always in the form of 80-byte fixed-length records, blocked or unblocked. The compiler always reserves two buffers of 80 bytes each; however, you may specify a block size of more than 80 bytes, provided that sufficient main storage is available to the compiler. (For further details see the discussion of the SIZE compiler option under "SIZE Option" on page 26.) The data set defined by the DD statement with the name SYSPUNCH is also used to store the output from the preprocessor if you specify the MDECK compiler option.

TEMPORARY WORKFILE (SYSUT1)

The compiler requires a data set for use as a temporary workfile. It is defined by a DD statement with the name SYSUT1, and is known as the spill file. It must be on a direct-access device, and must not be allocated as a multi-volume data set. The spill file is used as a logical extension to main storage and is used by the compiler and by the preprocessor to contain text and dictionary information.

The record size used depends on the amount of storage available to the compiler and whether or not the storage device is a 3330, 3340, 3350, or 3380.

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

Statement Lengths

The optimizing compiler has a restriction that any statement must fit into the compiler's work area. The maximum size of this work area varies with the amount of space available to the compiler. The maximum length of a statement is 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 in the same way the INITIAL attribute when it is followed by a list of items separated by commas that are not contained within parentheses. Each item may 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.

It is possible that programs with large DECLARE statements will not compile successfully on the optimizing compiler although they had compiled successfully on another compiler. The following techniques are suggested to overcome this problem:

- Increase the main storage available to the compiler, unless it already exceeds 128K bytes.
- 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');
```

LISTING (SYSPRINT)

The compiler generates a listing that includes all the source statements that it processed, information relating to the object module, and, when necessary, messages. Most of the information included in the listing is optional, and you can specify those parts that you require by including the appropriate compiler options. The information that may appear, and the associated compiler options, are described under "Compiler Listing" on page 46.

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

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

The compiler always reserves 258 bytes of main storage (129 bytes each) for two buffers for this data set; however, you may specify a block size of more than 129 bytes, provided that sufficient main storage is available to the compiler. (For further details of the SIZE compiler option, see "SIZE Option" on page 26.)

SOURCE STATEMENT LIBRARY (SYSLIB)

If you use the preprocessor %INCLUDE statement to introduce source statements into the PL/I program from a library, you can either define the library in a DD statement with the name SYSLIB, or you can choose your own ddname (or ddnames) and specify a ddname in each %INCLUDE statement. (For further information on the preprocessor, see "Compile-Time Processing (Preprocessing)" on page 59.)

EXAMPLE OF COMPILER JCL

A typical sequence of job control statements for compiling a PL/I program is shown in Figure 4. The DECK and NOOBJECT compiler options, described below, have been specified to obtain an object module as a card deck only. Job control statements for link editing an object module in the form of a card deck are shown in Chapter 3, "The Linkage Editor and the Loader" on page 65.

```
//OPT4#4 JOB
//STEP     EXEC  PGM=IEL0AA,PARM='DECK,NOOBJECT'
//SYSPUNCH DD SYSOUT=B
//SYSUT1 DD UNIT=SYSDA,SPACE=(1024,(60,60),,CONTIG)
//SYSPRINT DD SYSOUT=A
//SYSIN DD *
/*
```

Figure 4. Job Control Statements for Compiling a PL/I Program Not Using Cataloged Procedures.

COMPILER OPTIONS

The compiler provides a number of options, both at compile time and at execution time. Options that can be specified at compile time are known as compiler options. Options that can be specified at execution time are known as execution-time options.

Compiler options, their abbreviated syntax, and their defaults (as supplied by IBM) are shown in Figure 5 on page 14 and Figure 6 on page 16. An installation can modify defaults or delete options according to local requirements; check for any modified defaults at your installation. Deleted compiler options can be reinstated for a compilation by means of the CONTROL compiler option.

Also provided is the ability to pass an argument to the PL/I main procedure. This facility is described under "Specifying Execution-Time Options and Main Procedure Parameters in the EXEC Statement" on page 30.

SPECIFYING COMPILER OPTIONS

For each compilation, the IBM or installation default for a compiler option will apply unless it is overridden by specifying the option in a PROCESS statement or in the PARM parameter of an EXEC statement.

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

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 COMPILER 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:

```
//STEP1 EXEC PGM=IEL0AA,PARM='OBJECT,LIST'
```

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

```
//STEP1 EXEC PGM=IEL0AA,PARM=('AG,A',
//         'C,ESD,F(I),FLOW(10,1)',
// 'M,MI(''X''),NEST,STG,X')
```

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

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

SPECIFYING COMPILER OPTIONS IN THE *PROCESS STATEMENT

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

```
* PROCESS options;
```

where "options" is a list of compiler options. The list of options must be terminated with a semicolon and should not extend beyond the default right-hand source margin. The asterisk must appear in the first data byte of the record. If the records are F format, the asterisk must be in column 1. If the records are V or U format, the asterisk must be as far left as possible, that is column 1 if possible, or immediately following the sequence numbers if these are on the extreme left. The keyword PROCESS may follow in the next byte (column) or after any number of blanks. Option keywords must be separated by a comma and/or at least one blank.

Blanks are permitted before and after any non-blank delimiter in the list, with the exception of strings within quotation marks, for example MARGINI('*'), in which padding blanks should not be inserted.

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

```
* PROCESS;
```

Should it be necessary to continue the PROCESS statement onto the next card or record, terminate the first part of the list after any delimiter, up to the default right-hand margin, and continue on the next card or record. Option keywords or keyword arguments may be split, if required, 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. A PROCESS statement may be continued in several statements, or a new PROCESS statement started. For use of the PROCESS statement with batched compilation, see "Batched Compilation" on page 55.

COMPILER OPTION TYPES

The compiler options are of the following types:

1. Simple pairs of keywords: a positive form (for example, NEST) that requests a facility, and an alternative negative form (for example, NONEST) that rejects that facility.

2. Keywords that permit 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)).

The following paragraphs describe the options in alphabetic 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 "Compiler Listing" on page 46.

Figure 5 lists all the compiler options with their abbreviated syntax and their default values for batch mode. Defaults under TSO and CMS are given in the TSO User's Guide, and CMS User's Guide, respectively for this compiler.

Figure 6 on page 16 lists the options by function so that you can, for example, determine the preprocessing.

Compiler Option	Abbreviated Name	IBM Default
AGGREGATE|NOAGGREGATE	AG|NAG	NOAGGREGATE
ATTRIBUTES[(FULL|SHORT)]| NOATTRIBUTES	A[(F|S)]|NA	NOATTRIBUTES [(FULL[1])]
CHARSET([48|60][EBCDIC|BCD])	CS([48|60][EB|B])	CHARSET(60 EBCDIC)
COMPILE|NOCOMPILE[(W|E|S)]	C|NC[(W|E|S)]	NOCOMPILE(S)
CONTROL('password')	-	-
COUNT|NOCOUNT	CT|NCT	NOCOUNT
DECK|NODECK	D|ND	NODECK
ESD|NOESD	-	NOESD
FLAG[(I|W|E|S)]	F[(I|W|E|S)]	FLAG(I)
FLOW[(n,m)]|NOFLOW	-	NOFLOW
GONUMBER|NOGONUMBER	GN|NGN	NOGONUMBER
GOSTMT|NOGOSTMT	GS|NGS	NOGOSTMT
GRAPHIC|NOGRAPHIC	-	NOGRAPHIC
IMPRECISE|NOIMPRECISE	IMP|NIMP	NOIMPRECISE
INCLUDE|NOINCLUDE	INC|NINC	NOINCLUDE
INSOURCE|NOINSOURCE	IS|NIS	INSOURCE
INTERRUPT|NOINTERRUPT	INT|NINT	NOINTERRUPT
LINECOUNT(n)	LC(n)	LINECOUNT(55)
LIST[(m[,n])]|NOLIST	-	NOLIST
LMESSAGE|SMESSAGE	LMSG|SMSG	LMESSAGE
MACRO|NOMACRO	M|NM	NOMACRO
MAP|NOMAP	-	NOMAP
MARGINI('c')|NOMARGINI	MI('c')|NMI	NOMARGINI
MARGINS(m,n[,c])	MAR(m,n[,c])	MARGINS(2,72) or MARGINS(10,100) (see text)
MDECK|NOMDECK	MD|NMD	NOMDECK
NAME('name')	N('name')	-
NEST|NONEST	-	NONEST
NUMBER|NONUMBER	NUM|NNUM	NONUMBER

Figure 5 (Part 1 of 2). Compiler Options, Abbreviations, and Defaults in Batch Mode

Compiler Option	Abbreviated Name	IBM Default
OBJECT|NOOBJECT	OBJ|NOBJ	OBJECT
OFFSET|NOOFFSET	OF|NOF	NOOFFSET
OPTIMIZE(TIME|0|2)| NOOPTIMIZE	OPT(TIME|0|2)NOPT	NOOPTIMIZE
OPTIONS|NOOPTIONS	OP|NOP	OPTIONS
SEQUENCE(m,n)|NOSEQUENCE	SEQ(m,n)|NSEQ	F-format: SEQUENCE(73,80) V-format: SEQUENCE(1,8)
SIZE([-]yyyyyyyy| [-]yyyyyK|MAX)	SZ([-]yyyyyyyy| [-]yyyyyK|MAX)	SIZE(MAX)
SOURCE|NOSOURCE	S|NS	SOURCE
STMT|NOSTMT	-	STMT
STORAGE|NOSTORAGE	STG|NSTG	NOSTORAGE
SYNTAX|NOSYNTAX[(W|E|S)]	SYN|NSYN[(W|E|S)]	NOSYNTAX(S)
TERMINAL[(opt-list)]| NOTERMINAL	TERM[(opt-list)]|NTERM	NOTERMINAL
XREF[(FULL|SHORT)]|NOXREF	X[(F|S)]|NX	NOXREF[(FULL[1])]

Figure 5 (Part 2 of 2). Compiler Options, Abbreviations, and Defaults in Batch Mode

Note to Figure 5:

[1] FULL is the default suboption if the suboption is omitted with ATTRIBUTES or XREF

LISTING OPTIONS Control listings produced	**AGGREGATE** list of aggregates and their size **ATTRIBUTES[(SHORT|FULL)]** list of attributes of identifiers **ESD** list of external symbol dictionary **INSOURCE** list of preprocessor input **FLAG(I|W|E|S)** suppress diagnostic messages below a certain severity **LIST** list compiled code produced by compiler **MAP** lists offsets of variables in static control section and DSAs **OPTIONS** list of options used **SOURCE** list of source program or preprocessor output **STORAGE** list of storage used **XREF[(SHORT|FULL)]** list of statements in which each identifier is used
Improve readability of source listing	**NEST** indicates do-group and block level by numbering in margin **MARGINI** highlights any source outside margins
Control lines per page of listing	**LINECOUNT** specifies number of lines per page on listing
INPUT OPTIONS	**GRAPHIC** specifies that graphics are used in source **CHARSET** identifies the character set used in source **MARGINS** identifies position of PL/I source and a carriage control character **SEQUENCE** specifies the columns used for sequence numbers
OPTIONS TO PREVENT UNNECESSARY PROCESSING	**NOSYNTAX(W|E|S)** stop processing after errors are found in preprocessing **NOCOMPILE(W|E|S)** stop processing after errors are found in syntax checking

Figure 6 (Part 1 of 2). Compiler Options Arranged by Function

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
OPTIONS TO IMPROVE PERFORMANCE	**OPTIMIZE/NOOPTIMIZE** improves execution performance but increases compilation time. NOOPTIMIZE does the reverse
OPTIONS TO USE WHEN PRODUCING AN OBJECT MODULE	**OBJECT** produce an object module from compiled output **NAME** specify the name of the object module produced **DECK** produce an object module in punched card format
OPTIONS TO CONTROL STORAGE USED	**SIZE** controls the amount of storage used by the compiler
OPTIONS TO IMPROVE USABILITY AT A TERMINAL	**TERMINAL** specifies how much of listing is transmitted to terminal **LMESSAGE/SMESSAGE** specifies concise or full message format
OPTIONS TO SPECIFY STATEMENT NUMBERING SYSTEM USED	**NUMBER & GONUMBER** numbers statements according to line in which they start **STMT & GOSTMT** numbers statements sequentially **OFFSET** specifies that a listing associating statement numbers with offsets will be generated
OPTIONS FOR USE WHEN DEBUGGING	**COUNT** generate code that, if execution-time COUNT is specified, will result in a count of the number of times each statement is executed **FLOW** generate code that, if execution-time FLOW is specified, will result in a trace of statements executed being retained
OPTION TO CONTROL EFFECT OF ATTENTION INTERRUPTS	**INTERRUPT** specifies that the ATTENTION condition will be raised after interrupt is caused

Figure 6 (Part 2 of 2). Compiler Options Arranged by Function

AGGREGATE OPTION

The AGGREGATE option specifies that the compiler is to include in the compiler listing an aggregate length table, giving the lengths of all arrays and major structures in the source program.

ATTRIBUTES [(FULL|SHORT)] OPTION

The ATTRIBUTES option specifies that the compiler is to include in the compiler listing a table of source-program identifiers and their attributes. If both ATTRIBUTES and XREF apply, the two tables are combined.

If SHORT is specified, unreferenced identifiers are omitted, making the listing more manageable.

If both ATTRIBUTES and XREF apply, and there is a conflict between SHORT and FULL, the usage is determined by the last option found. For example, ATTRIBUTES(SHORT) XREF(FULL) results in FULL applying to the combined listing.

The suboption default FULL means that FULL applies if the option is specified with no sub-option.

CHARSET OPTION

The CHARSET option specifies the character set and data code that you have used to create the source program. The compiler will accept source programs written in the 60-character set or the 48-character set, and in the Extended Binary Coded Decimal Interchange Code (EBCDIC) or Binary Coded Decimal (BCD).

60- OR 48-CHARACTER SET: If the source program is written in the 60-character set, specify CHARSET(60); if it is written in the 48-character set, specify CHARSET(48). The language reference manual for this compiler lists both of these character sets. (The compiler will accept source programs written in either character set if CHARSET(48) is specified, however, if the reserved keywords, for example, CAT or LE are used as identifiers, errors may occur.)

BCD OR EBCDIC: If the source program is written in BCD, specify CHARSET(BCD); if it is written in EBCDIC, specify CHARSET(EBCDIC). The language reference manual for this compiler lists the EBCDIC representation of both the 48-character set and the 60-character set.

If both arguments (48 or 60, EBCDIC or BCD) are specified, they may be in any order and should be separated by a blank or by a comma.

COMPILE OPTION

The COMPILE option specifies that the compiler is to compile the source program unless an unrecoverable error was detected during preprocessing or syntax checking. The NOCOMPILE option without an argument causes processing to stop unconditionally after syntax checking. With an argument, continuation depends on the severity of errors detected so far, as follows:

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

NOCOMPILE(E) No compilation if error, severe error, or unrecoverable error is detected.

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

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

COUNT OPTION

The COUNT option specifies (1) that the compiler is to produce code that, when the execution-time COUNT (or FLOW) option is specified, counts and lists the number of times each statement is executed, and (2) the the default execution-time option for COUNT|NOCOUNT be set to COUNT.

The COUNT option implies the GOSTMT option if the STMT option applies, or the GONUMBER option if the NUMBER option applies.

DECK OPTION

The DECK option specifies that the compiler is to produce an object module in the form of 80-column card images and store it in the data set defined by the DD statement with the name SYSPUNCH. Columns 73-76 of each card contain a code to identify the object module; this code comprises the first four characters of the first label in the external procedure represented by the object module. Columns 77-80 contain a 4-digit decimal number: the first card is numbered 0001, the second 0002, and so on.

ESD OPTION

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

FLAG OPTION

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

FLAG(I) List all messages.

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

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

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

FLOW OPTION

The FLOW option specifies (1) that the compiler is to produce code that, when the execution-time FLOW option is specified, lists the flow of control when the program is executed, and (2) that the default execution-time option for FLOW|NOFLOW be set to FLOW. The format of the FLOW option is:

FLOW[(n,m)]

where 'n' is the maximum number of entries to be included in the lists. It should not exceed 32767.

'm' is the maximum number of procedures for which the lists are to be generated. It should not exceed 32767.

The IBM default, if (n,m) is not specified, is (25,10).

The output produced by the FLOW option is described under "Execution-Time FLOW Option" on page 45.

GONUMBER OPTION

The GONUMBER option specifies that the compiler is to produce additional information that will allow line numbers from the source program to be included in execution-time messages. Alternatively, these line numbers can be derived by using the offset address, which is always included in execution-time messages, and the table produced by the OFFSET option. (The NUMBER option must also apply.)

Use of the GONUMBER option implies NUMBER, NOSTMT, and NOGOSTMT. If NUMBER applies, GONUMBER is forced by the COUNT option.

GOSTMT OPTION

The GOSTMT option specifies that the compiler is to produce additional information that will allow statement numbers from the source program to be included in execution-time messages.

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

Use of the GOSTMT option implies STMT, NONUMBER, and NOGONUMBER. If STMT applies, GOSTMT is forced by the COUNT option.

GRAPHIC OPTION

The GRAPHIC option specifies that either:

- You have graphics within comments in your source program
- You use the MACRO option and your source program contains graphics within comments or graphic constants

(You need not specify GRAPHIC if you use graphic constants and do not use the preprocessor.)

If you do not require graphic support, specify NOGRAPHIC. The default is NOGRAPHIC.

When using the GRAPHIC compiler option, ensure that all comments within your program use the hexadecimal value '0E' (or whatever value your installation has defined as the left delimiter) only as a left delimiter to begin a graphic string.

You must use the compiler option CHARSET=(EBCDIC,60) when the GRAPHIC compiler option is specified.

To print graphic data (including your source program), your data must be in a format acceptable for a printer with graphic support or for a print utility program such as the Kanji print utility.

IMPRECISE OPTION

The IMPRECISE option specifies that the compiler is to include extra text in the object module to localize imprecise interrupts when executing the program with an IBM System/360 Model 91 or an IBM System/370 Model 165 or 195. This extra text is generated for ON statements (to ensure that, if interrupts occur, the correct on-units will be entered), for null statements, and for ENTRY statements. The correct line or statement numbers will not necessarily appear in execution-time messages. If you need more accurate identification of the statement in error, insert null statements at suitable points in your program.

INCLUDE OPTION

The INCLUDE option requests the compiler to handle the inclusion of PL/I source statements for programs that use the %INCLUDE statement. For programs that use the %INCLUDE statement but no other PL/I preprocessor statements, this method is faster than using the preprocessor. If the MACRO option is also specified, the INCLUDE option has no effect.

INSOURCE OPTION

The INSOURCE option specifies that the compiler is to include a listing of the source program (including preprocessor statements) in the compiler listing. This option is applicable only when the preprocessor is used; therefore, the MACRO option must also apply.

INTERRUPT OPTION

This option determines the effect of attention interrupts when the compiled PL/I program is being executed under an interactive system. (If specified on a batch system, INTERRUPT may cause an abend.)

If INTERRUPT was in effect during compilation, an established ATTENTION on-unit will be executed when one attention interrupt is caused during execution. If there is no such on-unit, processing will continue.

If NOINTERRUPT was in effect during compilation, one attention interrupt during execution will end the execution of the program and cause control to return to the interactive system.

It should be noted that if any procedure within a load module was compiled with the INTERRUPT option, an attention interrupt will lead to the ATTENTION condition being raised if polling is carried out, and execution continuing with no apparent effect if polling is not carried out regardless of which option was used for the procedure in which the interrupt occurs. Polling is carried out during the execution of stream I/O for all modules, and, additionally, at branching points for modules compiled with the INTERRUPT option. Because the ATTENTION condition is raised when polling is done, an attention interrupt in a program partly compiled with the INTERRUPT option can lead to unexpected results.

LINECOUNT OPTION

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

```
LINECOUNT(n)
```

where 'n' is the number of lines. It must be in the range 1 through 32767, but only headings are generated if you specify less than 7.

LIST OPTION

The LIST option specifies that the compiler is to include a listing of the object module (in a syntax similar to assembler language instructions) in the compiler listing. The syntax of the LIST option is:

```
LIST[(m[,n])]
```

where 'm' is the number of the first, or only, source statement for which an object listing is required and '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 LIST is used in conjunction with MAP, additional listings of static storage are produced. (For further information on the MAP compiler option, see "MAP Option.")

LMESSAGE OPTION

The LMESSAGE and SMESSAGE options specify that the compiler is to produce messages in a long form (specify LMESSAGE) or in a short form (specify SMESSAGE).

MACRO OPTION

The MACRO option specifies that the source program is to be processed by the preprocessor.

MAP OPTION

The MAP option specifies that the compiler is to produce tables showing the organization of the static storage for the object module. A table showing the mapping of static and automatic variables with offsets from their defining bases is always produced. If the LIST option (described above) is also used, a map of the static internal and external control sections is also generated.

MARGINI OPTION

The MARGINI option specifies that the compiler is to include a specified character in the column preceding the left-hand margin, and in the column following the right-hand margin of the listings resulting from the INSOURCE and SOURCE options. Any text in the source input which precedes the left-hand margin will be shifted left one column, and any text that follows the right-hand margin will be shifted right one column. For variable-length input records that do not extend as far as the right-hand margin, the character is inserted in the column following the end of the record. Thus text outside the source margins can be easily detected.

The MARGINI option has the syntax:

```
MARGINI('c')
```

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

MARGINS OPTION

The MARGINS option specifies the part of each input record that contains PL/I statements. The compiler will not process data that is outside these limits (but it will include it in the source listings).

The option can also specify the position of an American National Standard (ANS) printer control character to format the listing produced if the SOURCE option applies. This is an alternative to using %PAGE and %SKIP statements (described in the language reference manual for this compiler). If you do not use either method, the input records will be listed without any intervening blank lines. The syntax of the MARGINS option is:

```
MARGINS(m,n[,c])
```

where 'm' is the column number of the leftmost character that will be processed by the compiler. It should not exceed 100.

'n' is the column number of the rightmost character that will be processed by the compiler. It should be greater than m, but not greater than 100.

'c' is the column number of the ANS printer control character. It should not exceed 100 and should be outside the values specified for m and n. 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.

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

The MARGINS option allows you to override the default for the primary input in a program. The secondary input must have either the same margins as the primary input if it is the same type of record, or default margins if it is a different type.

MDECK OPTION

The MDECK option specifies that the preprocessor is to produce a copy of its output (see also "MACRO Option" on page 22) and store it in the data set defined by the DD statement with the name SYSPUNCH. The last four bytes of each record in SYSUT1 are not copied, thus this option allows you to retain the output from the preprocessor as a deck of 80-column punched cards.

NAME OPTION

The NAME option specifies that the compiler is to place a linkage editor NAME statement as the last statement of the object module. When processed by the linkage editor, this NAME statement indicates that primary input is complete and causes the specified name to be assigned to the load module created from the preceding input (since the last NAME statement).

It is required if you want the linkage editor to create more than one load module from the object modules produced by batched compilation (see also "Batched Compilation" on page 55).

If you do not use this option, the linkage editor will use the member name specified in the DD statement defining the load module data set. You can also use the NAME option to cause the linkage editor to substitute a new load module for an existing load module with the same name in the library. The format of the NAME option is:

NAME('name')

where "name" has from one through eight characters, and begins with an alphabetic character. The linkage editor NAME statement is described in Chapter 3, "The Linkage Editor and the Loader" on page 65.

NEST OPTION

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

NUMBER OPTION

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

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

The position of the sequence field can be specified in the SEQUENCE option. Alternatively the following default positions are assumed:

- First 8 columns for undefined-length or variable-length source input records.
- Last 8 columns for fixed-length source input records.

These defaults are the positions used for line-numbers generated by TSO; thus it is not necessary to specify the SEQUENCE option, or change the MARGINS defaults, when using line numbers generated by TSO. Note that the preprocessor output has fixed-length records irrespective of the original primary input. Any sequence numbers in the primary input are repositioned in columns 73-80.

The line number is calculated 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 a sequence number is found that is not greater than the preceding line number, a new line number is formed by adding the minimum integral multiple of 100,000 necessary to produce a line number that is greater than the preceding one. If the sequence field consists only of blanks, the new line number is formed by adding 10 to the preceding one. The maximum line number permitted by the compiler is 134,000,000, or, when FLOW/COUNT is specified, the maximum becomes 33,000,000; numbers that would normally exceed this are set to this maximum value, Only eight digits are printed in the source listing; line numbers of 100,000,000 or over will be printed without the leading "1" digit.

If there is more than one statement on a line, a suffix is used to identify the actual statement in the messages. For example, the second statement beginning on the line numbered 40 will be 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.

OBJECT OPTION

The OBJECT option specifies that the compiler is to store the object module that it creates in the data set defined by the DD statement with the name SYSLIN.

OFFSET OPTION

The OFFSET option specifies that the compiler is to print a table of statement or line numbers for each procedure with their offset addresses relative to the primary entry point of the procedure. This information is of use in identifying the statement being executed when an error occurs and a listing of the object module (obtained by using the LIST option) is available. If GOSTMT applies, statement numbers, as well as offset addresses, will be included in execution-time messages. If GONUMBER applies, line numbers, as well as offset addresses, will be included in execution-time messages.

A method of determining statement or line numbers from the offsets given in error messages is given under "Statement Offset Addresses" on page 50.

OPTIMIZE OPTION

The OPTIMIZE option specifies the type of optimization required:

NOOPTIMIZE
: specifies fast compilation speed, but inhibits optimization for faster execution and reduced main storage requirements.

OPTIMIZE(TIME)
: specifies that the compiler is to optimize the machine instructions generated to produce a very efficient object program. A secondary effect of this type of optimization can be a reduction in 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.

OPTIMIZE(0)
: is the equivalent of NOOPTIMIZE.

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

The language reference manual for this compiler includes a full discussion of optimization.

OPTIONS OPTION

The OPTIONS option specifies that the compiler is to include in the compiler listing, a list showing the compiler options, to be used during this compilation. This list includes all those applied by default, those specified in the PARM parameter of an EXEC statement, and those specified in a PROCESS statement.

SEQUENCE OPTION

The SEQUENCE option specifies the extent of the part of each input line or record that contains a sequence number. This number is included in the source listings produced by the INSOURCE and SOURCE option. Also, if the NUMBER option applies, line numbers will be derived from these sequence numbers and will be included in the source listings in place of statement numbers. No attempt is made to sort the input lines or records into the specified sequence. The SEQUENCE option has the syntax:

SEQUENCE(m,n)

where '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 for fixed-length records is SEQUENCE (73,80); that for variable-length and undefined-length records is SEQUENCE (1,8).

If the SEQUENCE option is in effect, an external procedure cannot contain more than 32,767 lines. To be able to compile an external procedure containing more that 32,767 lines, the NOSEQUENCE option must be specified provided that the actual number of statements is no more than 32,767. Because NUMBER and NONUMBER imply SEQUENCE, these options also should not be specified.

SIZE OPTION

This option can be used to limit the amount of main storage used by the compiler. This is of value, for example, when dynamically invoking the compiler, to ensure that space is left for other purposes. The SIZE option can be expressed in five forms:

SIZE(yyyyyyyy)
: specifies that yyyyyyyy bytes of main storage are to be requested. Leading zeros are not required.

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

SIZE(-yyyyyy)
: specifies that the compiler is to obtain as much main storage as it can, and then release yyyyyy bytes to the operating system. Leading zeros are not required.

SIZE(-yyyK)
: specifies that the compiler is to obtain as much main storage as it can, and then release yyyK bytes to the operating system (1K=1024). Leading zeros are not required.

SIZE(MAX)
: specifies that the compiler is to obtain as much main storage as it can.

The IBM default is SIZE(MAX), which permits the compiler to use as much main storage in the partition or region as it can.

When a limit is specified, the amount of main 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.

The negative forms 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 regions of different sizes.

After the compiler has loaded its initial phases and opened all files, it attempts to allocate space for working storage.

If SIZE(MAX) is specified, it obtains all space remaining in the region or partition (after allowance for subsequent data management storage areas). 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 the region or partition is too small, that is, less than 128K bytes, or if too much space for buffers has been requested.

The value 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 established when the compiler is invoked cannot be changed for later programs in the batch. Thus it is ignored if specified in a PROCESS statement.

In a TSO environment, an additional 10K to 30K bytes must be allowed for TSO. The actual size required for TSO depends on which routines are placed in the link-pack area (a common main storage pool available to all regions).

For details on the use of the SIZE option under CMS, see the CMS User's Guide for this compiler.

SMESSAGE OPTION

See "LMESSAGE Option" on page 22.

SOURCE OPTION

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

STMT OPTION

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

STORAGE OPTION

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

SYNTAX OPTION

The SYNTAX option specifies that the compiler is to continue into syntax checking after initialization (or after preprocessing if the MACRO option applies) unless an unrecoverable error is detected. The NOSYNTAX option without an argument causes processing to stop unconditionally after initialization (or preprocessing). With an argument, continuation depends on the severity of errors detected so far, as follows:

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

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

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

If the SOURCE option applies, the compiler will generate a source listing even if syntax checking is not performed.

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

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

TERMINAL OPTION

The TERMINAL option is applicable only in a conversational environment. It specifies that a subset of or all of the compiler listing produced during compilation is to be printed at the terminal. If TERMINAL is specified without an argument, diagnostic and informatory messages are printed at the terminal. You can add an argument, which takes the form of an option list, to specify other parts of the compiler listing that are to be printed at the terminal.

The listing at the terminal is independent of that written on SYSPRINT. However, if SYSPRINT is associated with the terminal, only one copy of each option requested will be printed even if it is requested in the TERMINAL option and also as an independent option. The following option keywords, their negative forms, or their abbreviated forms, can be specified in the option list:

AGGREGATE, ATTRIBUTES, ESD, INSOURCE,
LIST, MAP, OPTIONS, SOURCE, STORAGE,
and XREF.

If the option does not apply to the SYSPRINT listing, specifying it in the TERMINAL option has no effect. The other options that relate to the listing (that is, FLAG, GONUMBER, GOSTMT, LINECOUNT, LMSESSAGE/SMESSAGE, MARGINI, NEST, NUMBER, and the SHORT and FULL suboptions of ATTRIBUTES and XREF) will be the same as for the SYSPRINT listing.

XREF [(SHORT|FULL)] OPTION

The XREF option specifies that the compiler is to include in the compiler listing a cross-reference table of names used in the program together with the numbers of the statements in which they are declared or referenced. For a description of the format and content of the cross-reference table, see "Cross-Reference Table" on page 48.

If the suboption SHORT is specified, unreferenced names are not listed.

The default suboption FULL means that FULL applies if the option is specified with no suboption.

If both XREF and ATTRIBUTES are specified, 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 FULL applying to the combined listing.

SPECIFYING EXECUTION-TIME OPTIONS

Each execution of a PL/I program requires that values be established for a set of PL/I execution-time options. These options determine many of the properties of a PL/I program's execution, including its performance, its error-handling characteristics, and its production of debugging and tuning information.

Generally, it is unwise to rely on default settings (whether IBM-supplied or supplied by your local system programming staff). Inappropriate settings of these options can adversely affect both the function and the performance of your program.

The correct settings of these options should be established for all PL/I programs that you execute on a production basis.

You should understand in particular that almost no action you can take can do more to optimize the performance of a PL/I program than the correct setting of these options. Conversely, inappropriate settings of them can seriously degrade the performance of even a well-coded PL/I program.

It is a waste of time to undertake serious performance measurement or performance-oriented modification of a PL/I program until the execution-time options have been set appropriately.

This fact is not new with Release 5; it is true of all prior releases of the OS PL/I Optimizing Compiler and Libraries as well.

If you are already aware of the importance of these options, and have already undertaken to establish the proper value for ISASIZE, for example, for some or all of your programs, then you should take note of the fact that Release 5 adds three new options related to storage management: ISAINC, HEAP, and TASKHEAP. These options are described below along with the other options provided prior to Release 5.

In most cases, a setting of ISASIZE which resulted in efficient execution of your PL/I program on Release 4 will continue to do so on Release 5, although this should be verified for programs the performance of which is of critical importance.

If proper execution-time options are being determined for the first time for a program, if the program is to exploit 31-bit addressing, or if the program is one which exhibits widely varying storage requirements depending on its input data, then the new storage-related execution-time options should be taken into account. (See the section below entitled, "Execution-Time Storage Requirements".)

For each execution, the IBM or installation default for an execution-time option will apply unless it is overridden by a PLIXOPT string in the source program or by the PARM parameter of the EXEC statement for the execution step.

An option specified in the PLIXOPT string overrides the default value, and an option specified in the PARM parameter overrides that specified in the PLIXOPT string.

When execution-time options are not passed as parameters at execution time, the ISA is acquired and used instead of internal work areas. This provides faster execution but adds the requirement that enough storage be available for the ISA. If any execution options are passed at execution time, execution will be slower.

SPECIFYING EXECUTION-TIME OPTIONS IN THE PLIXOPT STRING

Execution-time options can be specified in a source program by means of the following declaration:

```
DCL PLIXOPT CHAR(len) VAR INIT('strg')
            STATIC EXTERNAL;
```

where "strg" is a list of options separated by commas or blanks, and "len" is a constant equal to or greater than the length of "strg." The maximum length of "strg" is 250 characters.

If more than one external procedure in a job declares PLIXOPT as STATIC EXTERNAL, only the first string will be link-edited and available at execution time.

The PLIXOPT string is ignored in a Checkout Compiler/Optimizing Compiler mixture environment.

	OPTION (default underlined)	USE	Applies to Release
Storage Control	HEAP[1]	Control storage for allocated variables.	5
	ISAINC[1]	Size of increments of storage added to initial allocation.	5
	ISASIZE	Control initial allocation of working storage.	4 and 5
	REPORT|<u>NOREPORT</u>	Generate report of storage usage.	4 and 5
	TASKHEAP[1]	Control HEAP storage for each subtask for multitasking.	5
Debugging	COUNT[2]|NOCOUNT	List number of times each statement is executed.	4 and 5
	FLOW(n,m)[2]|NOFLOW	List last n branches and m changes of procedure.	4 and 5
Error Handling	<u>SPIE</u>|NOSPIE	Allow program check interrupts to be handled by PL/I (SPIE) or passed to system (NOSPIE).	4 and 5
	<u>STAE</u>|NOSTAE	Allow ABENDS to be handled, if possible, by PL/I (STAE), or by system (NOSTAE).	4 and 5

Figure 7. Execution Time Options Listed by Function

Notes to Figure 7:

1 May be used only if all of the application is Release 5.

2 Only works if the FLOW or COUNT option was specified at compile time. Default is what was specified at compile time.

SPECIFYING EXECUTION-TIME OPTIONS AND MAIN PROCEDURE PARAMETERS IN THE EXEC STATEMENT

The method of coding the PARM parameter in an EXEC statement is described under "Specifying Compiler Options in the EXEC Statement" on page 12.

If you are using a cataloged procedure, you must qualify the keyword PARM with the name of the execution step; for example:

```
//STEP   EXEC   PLIXCLG,
//              PARM.GO='ISASIZE(10K)'
```

You can also use the PARM field to pass an argument to the PL/I main procedure. To do so, place the argument, preceded by a slash, after the execution-time options. For example:

```
//GO   EXEC  PGM=OPT,
//           PARM='ISASIZE(10K)/ARGUMENT'
```

If you wish to pass an argument without specifying options, it should be preceded by a slash. For example:

```
//GO   EXEC  PGM=OPT,PARM='/ARGUMENT'
```

If you omit the slash, your program may execute correctly, but it will incur extra overhead and cause a message regarding "invalid options" to be sent to SYSPRINT.

The method of coding the PARM parameter in an EXEC statement is given under "Specifying Compiler Options in the EXEC Statement" on page 12. See also "Execution-Time Options."

EXECUTION-TIME OPTIONS

The following paragraphs describe the execution-time options, which can be specified in the EXEC statement or in the PLIXOPT string. The values of all parameters are filled in successively from the system defaults, the PLIXOPT string, and the PARM parameter of the EXEC statement. Figure 7 on page 30 lists the options by function.

COUNT
: specifies that a count is to be kept of the number of times each statement in the program is executed and that the results are to be printed when the program terminates. This option is discussed in greater detail under "Execution-Time COUNT Option" on page 44.

NOCOUNT
: specifies that statement counting is not to be performed.

FLOW[(n,m)]
: specifies that a list of the most recent transfers of control in the execution of the program is to be generated. This option is discussed in greater detail under "Execution-Time FLOW Option" on page 45.

NOFLOW
: specifies that a flow list is not to be produced.

HEAP
: **Release 5 Only)** separates storage for allocated (that is, CONTROLLED and dynamically allocated BASED) variables from all other PL/I storage and specifies how that storage is to be handled. In a multitasking environment, HEAP option values apply only to the major task; subtask allocated storage is governed by the TASKHEAP option. The HEAP option is discussed in greater detail under "Execution-Time HEAP Option (Release 5 Only)" on page 36.

  The HEAP option has four parameters. These include one or two positional parameters, both optional, which must be numeric. If one or more of the positional parameters is omitted, then one or two keyword parameters can still be specified. No leading commas are required to specify only the keyword parameters. If the second positional parameter is specified but the first omitted, then a leading comma would be required to indicate the missing first positional parameter.

The syntax of the HEAP option is:

HEAP(size,increment,ANYWHERE|BELOW,KEEP|FREE)

where:

size — is optional. If specified, it determines the minimum initial size of heap storage, and is specified in bytes or as nnnK or as nnM. Storage is acquired in multiples of 4K. If not specified, no heap area is used. The IBM-supplied default is HEAP(0), that is, the HEAP option is not in effect.

increment — is optional. If specified, it determines the minimum size of any subsequent increment to the heap area. Storage is acquired in multiples of 4K. The IBM-supplied default value for the HEAP increment is 4K.

ANYWHERE — specifies that PL/I can allocate the heap area anywhere in storage. In an MVS/XA environment, this allows PL/I to locate heap storage either above or below 16 megabytes; PL/I will usually place it above 16 megabytes. In a non-MVS/XA environment, use of ANYWHERE necessarily places heap storage below 16 megabytes. ANYWHERE is the IBM-supplied default.

BELOW — specifies that PL/I must allocate heap storage below 16 megabyte, in storage accessible to 24-bit addressing.

KEEP — specifies that storage allocated to HEAP increments will not be released when a FREE statement in the program deallocates the last variable stored there. This is the IBM-supplied default.

FREE — specifies that storage allocated to HEAP increment will be released when the last variable occupying it is FREEd.

ISAINC — **(Release 5 Only)** specifies the minimum size of an increment to the ISA.

If ISAINC is not specified, when the storage currently allocated to the ISA is not large enough to handle all of a program's storage requests, only that amount of storage needed at the time of the request is obtained. When ISAINC is used, the amount of storage allocated when the ISA is too small for the current request is the larger of the ISAINC size or the requested size, rounded up to the next higher multiple of 4K. Thus the use of the ISAINC option can save the increased execution time caused by frequent GETMAINS of small amounts of storage.

The syntax of the ISAINC option is:

```
ISAINC(size1,size2)
```

where:

size1 — specifies the minimum amount by which the ISA for the major task will be incremented, and is specified in bytes or as nnnk or nnM. The IBM-supplied default is ISAINC=0.

size2 — specifies the minimum amount by which the ISA for any subtask will be incremented, and is specified in bytes or as nnnk or nnM. "size2" is ignored in a nontasking environment.

ISASIZE — specifies the storage sizes and number of subtasks.

The syntax of the ISASIZE option is:

```
ISASIZE(size1,size2,tasks)
```

where:

size1 — specifies the length of the initial storage area.

This specifies the main (or only) task size, in bytes or as nnnK or as nnM. It can be preceded by a minus sign. The storage will be contiguous.

A size of '0' causes PL/I to issue a GETMAIN request for the largest block of contiguous storage in the region; PL/I then returns half of that block to the system and retains the other half as its ISA.

The minus sign is used when stating the amount of storage in the region or partition that must be left outside the resident load module and the ISA. This storage will be contiguous. A value of '-0' should not be specified unless the largest possible ISA is required and no files, including SYSPRINT, will be used, and no subtasks may be allocated. Otherwise an ABEND may occur because of lack of system storage.

ISASIZE=0 is the IBM-supplied default in a nontasking environment. In a multitasking environment, the default is 8192 bytes.

size2 — specifies the length of each subtask initial storage area. This is an unsigned integer, n bytes, nnnK, or nnM.

"size2" is ignored in a nontasking environment.

tasks — is a decimal integer that is the maximum number of subtasks. The IBM-supplied default is 20.

"tasks" is ignored in a nontasking environment.

Commas must be provided if "size2" or "tasks" is used and earlier arguments are omitted.

REPORT — specifies that a report of the use of storage by a program will be generated and placed on the file with the ddname PLIDUMP or PL1DUMP at the end of execution. A description of the output and how to make use of it is given in "Execution-Time Storage Requirements for Nonmultitasking Programs" on page 37.

REPORT output is headed by the name of the main procedure and the time and date of the end of execution. You can also supply your own identifier using the PLIXHD string. For more information on PLIXHD, see "Using PLIXHD to Identify COUNT and REPORT Output" on page 37.

If no DD statement is provided for PLIDUMP or PL1DUMP, a message is generated and the report is not given.

The use of the REPORT option downgrades performance.

NOREPORT — specifies that no program management report is required. This option may be abbreviated to NR.

SPIE — specifies that when a program interrupt occurs, the PL/I error handler is to be used. Under certain circumstances the ERROR condition will be raised.

NOSPIE — specifies that on program initialization, PL/I will not issue the SPIE or ESPIE macro to request control after a program check. Unless running under MVS/XA, do not use NOSPIE when extended precision variables are used in the PL/I source program.

STAE — specifies that when an ABEND occurs, the PL/I library routines are to attempt to raise the ERROR conditions or to produce a diagnostic message and a PLIDUMP.

NOSTAE — specifies that on program initialization, PL/I will not issue the STAE or ESTAE macro to request control after an ABEND.

TASKHEAP — **(Release 5 Only)** specifies that a separate heap storage area is to be created for each subtask in a multitasking environment. This separates storage for CONTROLLED and dynamically allocated BASED variables in a subtask from all other PL/I storage and specifies how that storage is to be handled.

The syntax of the TASKHEAP option is:

TASKHEAP(size,increment,ANYWHERE|BELOW,KEEP|FREE)

where:

size — is optional. If specified, it determines the minimum initial size of taskheap storage, and is specified in bytes or as nnnK or as nnM. Storage is acquired in multiples of 4K. If not specified, no taskheap area is used. The IBM-supplied default is TASKHEAP(0), that is, the TASKHEAP option is not in effect.

increment — is optional. If specified, it determines the minimum size of any subsequent increment to the taskheap areas. Storage is acquired in multiples of 4K. The IBM-supplied default value for the TASKHEAP increment is 4K.

ANYWHERE — specifies that PL/I can allocate the taskheap areas anywhere in storage. In an MVS/XA environment, this allows PL/I to locate taskheap storage either above or below 16 megabytes; PL/I will usually place it above 16 megabytes. In a non-MVS/XA environment, use of ANYWHERE necessarily places taskheap storage below the line. ANYWHERE is the IBM-supplied default.

BELOW — specifies that PL/I must allocate taskheap storage below 16 megabytes, in storage accessible to 24-bit addressing.

KEEP — specifies that storage allocated to TASKHEAP increments will not be released when a FREE statement in the program deallocates the last variable stored there. This is the IBM-supplied default.

FREE — specifies that storage allocated to TASKHEAP increments will be released when the last variable occupying it is FREEd.

EXECUTION-TIME ISASIZE OPTION

The types of information kept in the ISA vary depending on whether or not the HEAP option applies to the execution of your current PL/I program.

The values you specify for ISASIZE and for the related ISAINC, HEAP, and TASKHEAP options determine the method used to acquire storage for your program and, consequently, the time and space that it uses.

It is important to set these values appropriately for each PL/I program. Appropriate values for ISASIZE and ISAINC can significantly reduce the number of GETMAINs and FREEMAINs required for execution of your PL/I program.

Your major source of input for proper specification of options generally is the PL/I storage management report, produced using the REPORT option.

Storage associated with the ISA (and increments to the ISA) is acquired below 16 megabytes on MVS/XA, so it is always addressable in 24-bit mode.

On MVS/XA, since the ISA resides below 16 megabytes, the residual storage requested by a negative value of ISASIZE is residual storage below 16 megabytes.

Note that the load module containing the PL/I program is not always located in the user's region below 16 megabytes. For example, the load module may be loaded above 16 megabytes on MVS/XA, or it can be located in the link pack area of the operating system.

EXECUTION-TIME ISAINC OPTION (RELEASE 5 ONLY)

Whether or not the ISAINC option is used, both the ISA and all increments added to it reside in storage below 16 megabytes, so that the ISA and all increments to it are addressable in 24-bit addressing mode under MVS/XA.

EXECUTION-TIME HEAP OPTION (RELEASE 5 ONLY)

If the value of the initial heap allocation results in zero, then the HEAP option is not active. In this case, no separate heap area is utilized, and all storage goes into the ISA or into increments to the ISA. Such areas reside below 16 megabytes on MVS/XA.

The HEAP option performs these functions:

1. It separates storage allocated to PL/I variables which you allocate with PL/I ALLOCATE statements, (that is, CONTROLLED variables and dynamically allocated BASED variables), from all other PL/I storage. It causes such variables to be placed in a separate "heap" area, rather than in the ISA or an increment to the ISA. You may control both the minimum initial size of the heap area and the minimum sizes of subsequent increments to the heap area. You can improve performance by picking values for both HEAP and ISASIZE that will minimize the number of times PL/I must acquire storage from the operating system.

   Neither the original heap area nor any increment to it is acquired until your program executes an ALLOCATE statement which requires storage not currently available in the heap area.

   Each acquisition of storage for the heap area is in multiples of 4K bytes aligned on a 4K-byte page boundary. The first eight bytes of each such area contains PL/I housekeeping information. Thus a 4K-byte heap increment occupies 4096 bytes, but provides 4088 bytes of space to hold your data.

   PL/I will place as many of your CONTROLLED or dynamically allocated BASED variables in a unit of the heap area as will fit.

   A based variable requires no additional space beyond itself, although all allocations are begun on double-word boundaries. A controlled variable requires a PL/I control and possibly a PL/I string or aggregate descriptor in addition to the variable itself.

2. It allows you to specify whether PL/I should free an increment of heap storage when FREE statements issued by your program leave a unit of the heap area empty.

   The initial heap allocation is retained until program or task termination.

3. In the MVS/XA environment, it allows you to specify whether the heap area must be kept below 16 megabytes, or whether the heap area can go anywhere. If the latter is specified, and your program is being executed in 31-bit addressing mode, PL/I will normally put the heap area above 16 megabytes on MVS/XA. If you are executing your program in 24-bit addressing mode on MVS/XA or if you are executing your program on a non-MVS/XA system, either BELOW or ANYWHERE may be specified, but the heap area will necessarily be acquired in storage below 16 megabytes.

USING PLIXHD TO IDENTIFY COUNT AND REPORT OUTPUT

When COUNT or REPORT output is generated and your program contains a **static external** character variable called PLIXHD, the value in PLIXHD is printed at the head of the output after the name of the main procedure and the date and time of execution. This allows you to supply an identifier for such output.

To do this, PLIXHD must be declared as STATIC EXTERNAL CHARACTER VARYING. (STATIC may be omitted because all EXTERNAL data is STATIC by default). For example:

```
DCL PLIXHD EXTERNAL CHARACTER(50) VARYING
  INIT('THIS IS A PLIXHD MESSAGE')
```

The printed output of PLIXHD is limited to one line and is truncated if necessary. The result of using PLIXHD as shown above would be:

```
STORAGE MANAGEMENT REPORT FOR PROCEDURE P
DATE 26 NOVEMBER 1981 TIME 13.15.16.00
THIS IS A PLIXHD MESSAGE
(Report Output goes here)
```

If PLIXHD is declared EXTERNAL but not CHARACTER VARYING, a diagnostic message is generated during compilation. If PLIXHD is CHARACTER but not VARYING, its value is printed as shown above. In other cases, it will normally be ignored but could lead to execution time errors.

EXECUTION-TIME STORAGE REQUIREMENTS FOR NONMULTITASKING PROGRAMS

During the execution of a nonmultitasking program, the region used by your PL/I program is divided into three areas; the load module, the ISA (Initial Storage Area), and the remainder, called for convenience during the rest of this discussion residual storage. If you have used the HEAP execution-time option, a fourth area, heap storage, will be established in the residual area when your program uses the ALLOCATE statement. See Figure 8 on page 39.

The load module is used for the compiled code, constants, and storage for STATIC variables. The ISA is used for storage of all variables that are not STATIC and certain housekeeping fields. Heap storage is used for controlled and dynamically allocated BASED variables. These are referred to as PL/I storage. Residual storage is used for I/O buffers and transiently loaded routines from the PL/I and system libraries. It is also used as an overflow area for the ISA and heap and, consequently, may be used for PL/I storage.

The ISA is acquired by the PL/I program at the start of execution and retained until termination. Consequently, obtaining and freeing of storage within it can be managed by the PL/I program without resorting to system facilities. Thus the overheads of obtaining and freeing storage within the ISA are small compared with using the residual area where GETMAIN and FREEMAIN macro instructions have to be used. Execution is, therefore, faster if all PL/I storage is contained in the ISA. However, if significant parts of the ISA remain unused throughout long periods during the execution of a program, space

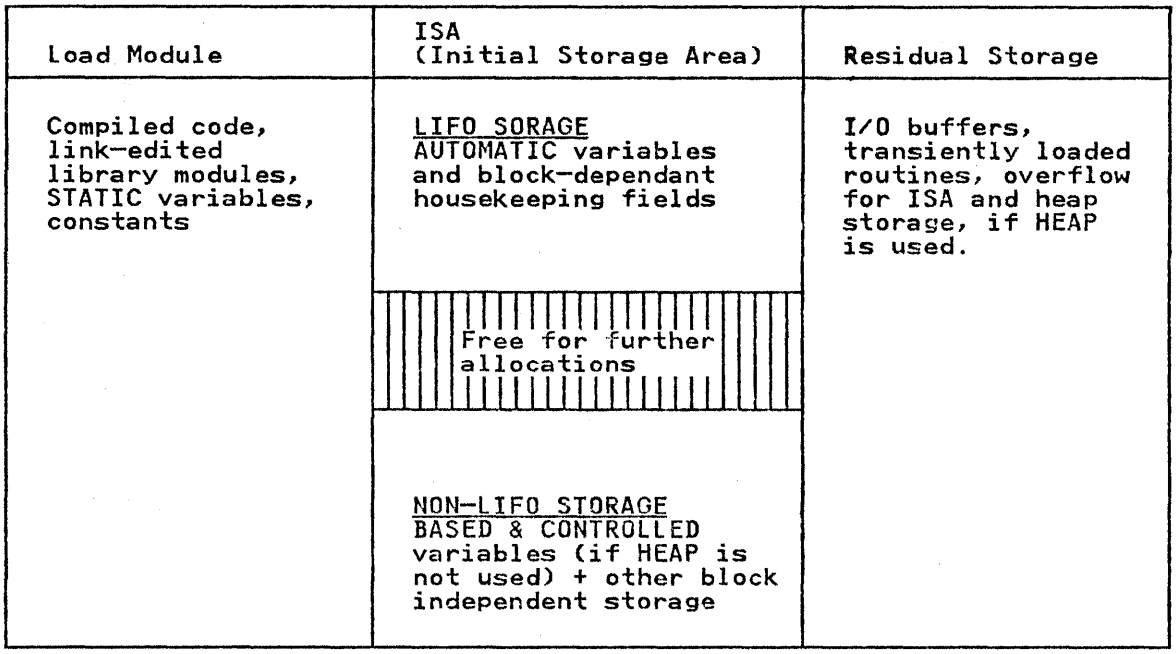
is wasted because storage within the ISA cannot be used for buffers or transient routines which must use the residual area. Appropriate choice of the value of ISAINC can help reduce system overheads if it is impractical to specify an ISASIZE large enough to hold all PL/I storage. The fact that ISA storage is quickly acquired and freed, but conversely may only be used for certain items makes the choice of ISA size a critical factor in determining both the time and space requirements of your program.

Heap storage is acquired when the first ALLOCATE is encountered during program execution. Increments to heap storage are obtained when there is not enough space in the existing heap storage to satisfy an ALLOCATE request and freed when all variables within the increment have been freed. The initial heap storage segment is retained until the main PL/I procedure terminates.

By using the REGION parameter in JCL in systems other than MFT, you can control the total size of the storage available to your program, and by using the ISASIZE execution time option you can control how much of the region is included in the ISA. Output from the REPORT option will indicate the best ISASIZE. This, together with installation accounting information, will help to determine the minimum practical region size.

When the REPORT option is in force, the use of storage is monitored and a report generated at the end of the program. The report is transmitted to the file with the ddname of PLIDUMP or PLIDUMP and is identified by the name of the main procedure and the date and time of execution. Optionally, the user can generate a further report identifier by use of PLIXHD. The REPORT option should only be used while the ISA size is being determined. It involves a considerable time overhead and should be removed as soon as possible. REPORT should be used after COUNT and FLOW have been removed, because COUNT and FLOW use extra storage and so make the report inaccurate.

LAYOUT OF REGION FOR NONMULTITASKING



LAYOUT OF REGION FOR MULTITASKING

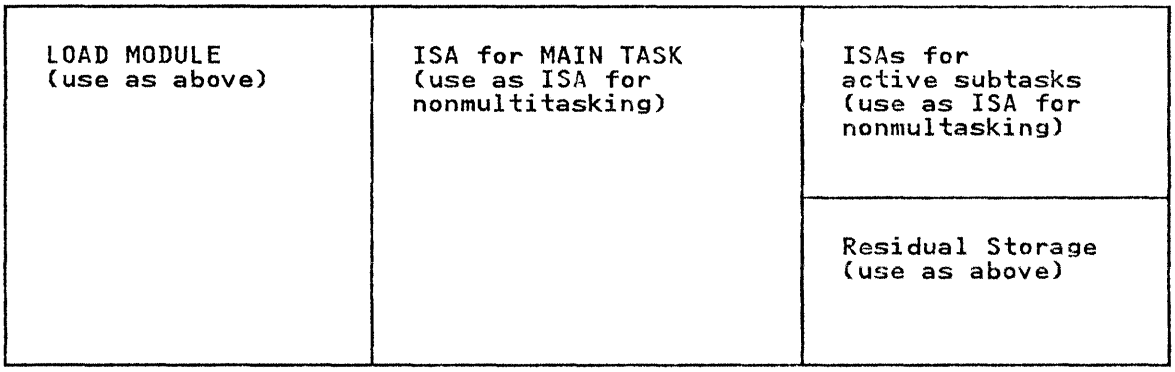


Figure 8. Storage Arrangements in Multitasking and Nonmultitasking Programs

USING THE REPORT OPTION

When using the REPORT option, the best strategy to ensure satisfactory results is to specify a very large ISASIZE so that the chances of all PL/I storage being within the ISA are high. This gives the most accurate estimate of PL/I storage used, and so the most accurate indication of the ISA size required. The ISA size should then be set to the size of the PL/I storage used and the program run again with the REPORT option to see if the ISA size is satisfactory. It should be born in mind that different data, or different paths through the program may result in different storage requirements. If it is impractical to specify a large ISA, an alternative is to specify a value of 1 and an ISAINC value of 0. This results in the minimum acceptable ISASIZE being used. This minimum is such that PL/I storage for the first and all subsequent blocks will be met from residual storage. The disadvantage of this method is that it tends to slightly overestimate the total amount of PL/I storage used. Because of the method of measurement used, an ISASIZE where PL/I storage is partly inside and partly outside the ISA gives the least satisfactory result.

The output caused by the REPORT option for a nonmultitasking program is shown with explanatory notes in Figure 9 on page 41.

```
STORAGE MANAGEMENT REPORT FOR MAIN PROCEDURE TEST
DATE 13 AUG 84  TIME 16.59.13.00

ISASIZE SPECIFIED 102400 BYTES  The size specified in the
                                ISASIZE option.  If the option
                                is not used, for nonmultitasking,
                                0 is given.  For multitasking, 8192
                                bytes is given.

ISAINC SPECIFIED 0 BYTES        The size specified in the
                                ISAINC option.   If this option
                                is not used, 0 is given.

LENGTH OF INITIAL STORAGE AREA (ISA) 102400 BYTES
                                Length used.
                                Normally this is the length specified
                                or the default (half of what's left
                                when the load module is loaded.)  However,
                                if this is not large enough for the
                                requirements of the first block,
                                another value is used.

AMOUNT OF PL/I STACK STORAGE REQUIRED 3074048 BYTES
                                This is the maximum amount of storage
                                that could have used the ISA.
                                It is the optimum ISASIZE in most
                                conditions but see text for provisos.

AMOUNT OF STORAGE OBTAINED OUTSIDE ISA 3074048 BYTES
                                Overflow of ISA, if any.  0 means none.

NUMBER OF STACK GETMAINS 3      Number of times ISA overflowed.
NUMBER OF STACK FREEMAINS 0     Number of times ISA overflow was freed.

NUMBER OF GET NON-LIFO REQUESTS 4
                                Number of times non-LIFO storage was
                                requested.
NUMBER OF FREE NON-LIFO REQUESTS 1
                                Number of times freeing of non-LIFO
                                storage was requested.
                                Non-LIFO storage is storage that is
                                not attached to a block, for example,
                                BASED and CONTROLLED storage, as opposed
                                to AUTOMATIC storage that is.
                                For a full description, see the
                                Execution Logic Manual.

HEAP SIZE SPECIFIED 0 BYTES     The size specified in the HEAP option.
                                If the option is not specified, 0 is given.

HEAP INCREMENT SPECIFIED 4096 BYTES
                                The minimum size of subsequent increments
                                to HEAP storage, specified in a HEAP
                                option parameter.  If the parameter is
                                not used, 4K is given.

AMOUNT OF PL/I HEAP STORAGE REQUIRED 0 BYTES
                                This is the maximum amount of storage that
                                heap could have used.

NUMBER OF HEAP GETMAINS 0       Number of times heap overflowed.

NUMBER OF HEAP FREEMAINS 0      Number of times heap overflow was freed.

NUMER OF GET HEAP REQUESTS 0    Number of times heap storage was requested.
NUMBER OF FREE HEAP REQUESTS 0  Number of times freeing of heap storage was
                                requested.
```

Figure 9. REPORT Output and Its Meaning (Release 5 Example)

Figure 9 on page 41 shows the output from the REPORT option. An ISA size equal to the "Amount of PL/I Storage Required" value in the report will give the fastest execution time, because it will allow all PL/I storage to be obtained within the ISA. However, it may increase overall size requirements, for example, if a program uses large BASED or CONTROLLED variables for a short time during execution when HEAP is not used, or if a little used subroutine contains a number of large variables, use of an ISASIZE equal to the "PL/I Storage Required" figure may be uneconomic as it will lead to the need for an unnecessarily large region. Where space is critical, increase of ISA size without increasing the REGION size may lead to the program terminating because of lack of space.

The most important line items on the report other than "Amount of PL/I Storage Required" are those which specify numbers of GETMAIN and FREEMAIN requests. Those associated with the ISA and its increments are identified as "stack" GETMAIN's and FREEMAIN's. Those associated with the HEAP area are identified as "heap" GETMAIN's and FREEMAIN's. These counts are important because they show the cost associated with non-optimal ISASIZE and HEAP values. If the size of the ISA can be cut in half at a cost of a few extra GETMAIN and FREEMAIN requests, then that may be acceptable or even desirable in some circumstances. If the cost is thousands or millions of extra GETMAIN and FREEMAIN requests, then it is probably unacceptable. The goal of the ISASIZE, ISAINC, and HEAP options is to permit a trade-off to be made between the amount of storage required and the cost of the GETMAIN and FREEMAIN requests required to manage storage.

If a program has to run in the smallest possible area, it is normally best to use an ISA size of 1. This results in all storage requests being made within the residual area, thus all spare storage is available for all purposes. This method does have a disadvantage, however, where a large number of small items, such as based variables, have to be allocated, because each item requires eight additional bytes for chaining.

When optimum sizes for ISASIZE, ISAINC, and HEAP have been determined, the program should be rerun with these sizes specified and the REPORT option still in force so that the results can be checked. When they are satisfactory the REPORT option should be removed.

FINDING THE OPTIMUM REGION SIZE

When the optimum storage options have been determined, the optimum region size can be determined using the System Management Facilities (SMF) of the system. These will tell you the region size used by your program. You should then specify the size used as the REGION size for subsequent runs. The SMF facilities are described in the operating system publications.

SMF does not give meaningful information about a PL/I program's use of storage unless a positive ISASIZE value is specified. If you want SMF storage data to be meaningful for a PL/I program, you should not let ISASIZE default to the IBM value of half the region excluding the load module, and you should avoid using a negative number for ISASIZE. The implementation of either of these values for ISASIZE requires that PL/I acquire the entire region via GETMAIN and then release part of it via FREEMAIN. The system accounting information provided by SMF in either case will always show the entire region being used. This is not useful for determining anything about the program's actual storage requirements, and it may cause inflated billing charges if SMF data is used to charge for storage.

EXECUTION-TIME STORAGE FOR MULTITASKING PROGRAMS

During the execution of a multitasking program, the region is divided into the load module area, an ISA for every task (each having the lifetime of its task), and the residual area that reabsorbs the ISA of a task when it is detached. See Figure 8 on page 39. The HEAP option, if in effect, provides a HEAP area for the major task. If it is desired to provide separate heap areas for the subtasks, then the TASKHEAP option can be specified to accomplish this. On MVS/XA the load modules (and thus STATIC storage) may reside above 16 megabytes, and the heap areas associated with the HEAP and TASKHEAP options may reside above or below 16 megabytes. The various types of storage are used for the same purposes as they are for nonmultitasking programs, except that ISAs of subtasks (and the TASKHEAP areas if they are required) are taken from the residual area, and later returned to the residual area when the subtask terminates.

You should review the discussion above concerning storage management for nonmultitasking. The various considerations discussed there concerning ISASIZE, ISAINC, and HEAP apply to ISASIZE, ISAINC, HEAP, and TASKHEAP for multitasking programs.

Every time a task is attached, an ISA is acquired. Because ISAs can only be used for certain types of storage, there is a danger of the free area for transient routines and other storage items that cannot use ISAs becoming too small. Consequently, the desirability of keeping all PL/I storage within the ISA is considerably reduced when compared with nonmultitasking programs.

USING THE REPORT OPTION

For multitasking programs, the REPORT option generates a report of storage use that can be used to determine the optimum size for the ISA of the main task, and the optimum size for the ISAs of all subtasks. It can in addition be used to evaluate the need for and effectiveness of values used for the ISAINC, HEAP, and TASKHEAP options. The report contains the information shown in Figure 9 on page 41 above for the main task, plus a combined listing for all subtasks containing the information shown below.

- Largest and smallest ISA sizes used by subtasks.
- Largest and smallest amounts of PL/I storage obtained by subtasks.
- Largest and smallest amounts of PL/I storage obtained outside the ISA as increments to the ISA by any subtask.
- Largest and smallest amounts of PL/I storage obtained as heap storage by any subtask, provided that the TASKHEAP option is active.
- Total number of GETMAIN and FREEMAIN requests issued by all subtasks to acquire and release increments to ISAs, identified as "stack" GETMAINs and FREEMAINs in the report.
- Total number of GETMAIN and FREEMAIN requests issued by all subtasks to acquire and release TASKHEAP areas, identified as "heap" GETMAINs and FREEMAINs in the report.
- Maximum number of subtasks attached at any one time.

As with nonmultitasking programs, the fastest execution will be achieved if all tasks obtain all their PL/I storage from within their own ISA. To achieve this result, the first figure in the ISASIZE option should be set to the amount of PL/I storage obtained for the main task, and second to the largest amount of PL/I storage obtained for any subtask. Whether or not this is practical depends on the number of tasks active at any one time, the difference in the storage usage of the subtasks, and the storage use within each task.

When an ISA size has been determined, a further run with the REPORT option should be tried to ensure that the expected results have been achieved. When they are satisfactory, the REPORT option should be removed.

The third argument to ISASIZE (maximum number of active tasks) is used to determine the number of subtask control blocks that will be allocated. This figure is not critical as far as storage use is concerned because the control blocks are not large. However, if the figure specified (or defaulted) is exceeded, execution will terminate. A generous figure should, therefore, be specified for this argument.

When the optimum storage options have been established, the optimum region size can be calculated using the System Management Facilities (SMF) of the system. See "Finding the Optimum Region Size" on page 42.

EXECUTION-TIME COUNT OPTION

Statement count information can be obtained at execution time only if one of the compiler options COUNT or FLOW was specified at compile time. For further details, see "COUNT Option" on page 19, and "FLOW Option" on page 19. If FLOW but not COUNT was specified at compile time, COUNT must be specified at execution to obtain count information. If COUNT was specified at compile time, count information will be produced unless NOCOUNT is specified at execution time.

Count information can be produced only when a statement number table exists. If COUNT is specified at compile time, a table is automatically produced. If only FLOW is specified at compile time, and COUNT is specified at execution time, then to obtain count information, GOSTMT or GONUMBER must also be specified at compile time.

Count output is written on the PLIDUMP file, or on the SYSPRINT file if no dump file is provided. The output has the following format:

```
PROCEDURE name
     FROM          TO          COUNT
       1           20            1
      21           30           10
       .            .            .
       .            .            .
      200          210           1
```

Three such columns are printed per page.

To draw attention to statements that have not been executed, ranges for which the count is zero are listed separately after the main tables.

The count tables are printed when the program terminates. If a procedure is invoked with one of the multitasking options, the count table for the invocation is printed when the task terminates.

Count output is headed by the name of the main procedure and the time and date the output was generated. You can also supply your own identifier for the output using the PLIXHD string. For more information on PLIXHD, see "Using PLIXHD to Identify COUNT and REPORT Output" on page 37.

If no DD statement is provided for PLIDUMP or PL1DUMP, a message is generated and COUNT output is written onto SYSPRINT if it has a suitable format.

Under CICS, COUNT output is sent to SYSPRINT; for further discussion see Chapter 15, "Using PL/I on CICS" on page 360.

If an invocation is terminated as a result of the termination of another task, its count table cannot be printed, because it is impossible to determine the point at which it terminated. In these circumstances, only the count table for the first task to terminate can be printed. For example, although a STOP statement will cause all tasks to be terminated, only the count table for the task containing the statement will be printed.

Count and flow output can be produced only for the main procedure and inner procedures compiled with it. When control is passed to a separate external PL/I procedure, any COUNT or FLOW options in force are suspended until control is returned to the main procedure. Only the compiler options that applied for compilation of the main procedure have any effect on execution-time COUNT and FLOW facilities.

EXECUTION-TIME FLOW OPTION

Flow information can be obtained at execution time only if one of the compiler options COUNT or FLOW was specified at compile time. For further details on these options, see "COUNT Option" on page 19, and "FLOW Option" on page 19. If FLOW was not specified at compile time, it must be specified at execution time to obtain flow information. If FLOW was specified at compile time, flow information will be produced unless NOFLOW is specified at execution time.

The format of the execution-time FLOW option is the same as that of the compile-time FLOW option, that is:

```
FLOW[(n,m)]
```

where 'n' is the maximum number of entries to be made in the flow output, and 'm' is the maximum number of procedures for which entries are to be made. Neither 'n' or 'm' may exceed 32,767.

If 'n' and 'm' are not specified at execution time, they are set as follows:

- If FLOW was specified or defaulted at compile time, the values of 'n' and 'm' specified or defaulted at compile time are used at execution time.
- If FLOW was specified at compile time without the subparameters (n,m), the IBM default values (25,10) are used.
- If NOFLOW was specified or defaulted at compile time, the IBM default values (25,10), are used.

Flow output is written on the SYSPRINT file whenever an on-unit with the SNAP option is executed. It is also included as part of PLIDUMP output if "T" is included in the dump options string.

The format of each line of flow output is:

```
sn1  TO  sn2  [IN name]
```

where:

sn1
: is the number of the statement from which the branch was made (the **branch out** point).

sn2
: is the number of the statement to which the branch was made (the **branch in** point).

name
: is the name of the procedure or the type of the on-unit that contains "sn2" if this is different from that containing "sn1."

The branches are listed in the order in which they occur. The last 'n' branch-in/branch-out point and the last 'm' procedures or on-units are listed. If more than 'm' procedures or on-units are entered in the course of 'n' branches, changes prior to the last 'm' procedures or on-units are indicated by printing "UNKNOWN" for "name."

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 defined by the DD statement with the name SYSPRINT (usually output to a printer). In a TSO environment, you can also request a listing at your terminal (using the TERMINAL option). The following description of the listing refers to its appearance on a printed page.

An example of the listing produced for a typical PL/I program is given in Appendix D, "Sample Program" on page 407.

The first part of Figure 6 on page 16 shows the components that can be included in the compiler listing. The rest of this section describes them in detail.

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

The listing comprises a small amount of standard information that always appears, together with those items of optional information specified or supplied by default. The listing at the terminal contains only the optional information that has been requested in the TERMINAL option.

HEADING INFORMATION

The first page of the listing is identified by the name of the compiler, the compiler version number, the time compilation commenced (if the system has the timer feature), and the date; this page, and subsequent pages are numbered.

The listing either ends with a statement that no errors or warning conditions were detected during the compilation, or with one or more messages. The format of the messages is described under "Messages" on page 54. If the machine has the timer feature, the listing also ends with a statement of the CPU time taken for the compilation and the elapsed time during the compilation; these times will differ only in a multiprogramming environment.

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 used for the compilation, including the default options, appears on the first page.

PREPROCESSOR INPUT

If both the options MACRO and INSOURCE apply, the input to the preprocessor is listed, 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 exactly as described for the compiler messages described under "Messages" on page 54.

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 will be spaced accordingly. %NOPRINT and %PRINT statements can be used to suppress and restart the printing of the listing.

If the option NUMBER applies, and the source program contains line numbers, these numbers are printed to the left of each line.

If the option STMT applies, the statements in the source program are numbered sequentially by the compiler, and the number of the first statement in the line appears to the left of each line in which a statement begins. If the source statements are generated by the preprocessor, columns 82-84 contain diagnostic information, as shown in Figure 15 on page 59.

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 or lines 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.

ATTRIBUTE TABLE

If an identifier is declared explicitly, the number of the DECLARE statement is listed. An undeclared variable is indicated by asterisks. (Undeclared variables are also listed in an error message.) The statement numbers of statement labels and entry labels are also given.

The attributes INTERNAL and REAL are never included; they can be assumed 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, only explicitly declared attributes are listed.

For an array, the dimension attribute is printed first; the bounds are printed as in the array declaration, but expressions are replaced by asterisks and structure levels other than base elements have their bounds replaced by asterisks.

For a character string or a bit string, the length, preceded by the word BIT or CHARACTER, is printed as in the declaration, but an expression is replaced by an asterisk.

If the SHORT suboption applies, unreferenced identifiers are not listed.

CROSS-REFERENCE TABLE

If the cross-reference table is combined 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 statement numbers for the outermost block are given first, followed on the next line by the statement numbers for the inner blocks.

The PL/I text is expanded and optimized to a certain extent before the cross-reference table is produced. Consequently, some names that may appear only once within a source statement may acquire multiple references to the same statement number. By the same token, other names may appear to have incomplete lists of references, while still others may have references to statements in which the name does not appear explicitly.

For example:

- Duplicate references may 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 each aggregate in the program is mapped. 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 bit offset for unaligned bit-string data is not given). 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 the aligned attribute is specified or inferred for a structure or its elements, the proper alignment requirements will be consistent with respect to other elements in the structure, even though the table does not obviously indicate the proper alignment relative to the beginning of the table.
- The length of each element.
- The total length of each aggregate, structure and sub-structure.

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 execution. 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 may 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 word "adjustable" or "defined" appears in the the "offset" column while "param" for parameter appears in the "element length" and/or "total length" columns. Because the length of an aggregate may not be known during compilation, padding information cannot be printed.

An entry for a COBOL mapped structure, that is, for a structure into which a COBOL record is read or from which a COBOL record is written, or for a structure passed to or from a COBOL program, has the word "COBOL" appended. Such an entry will appear only if the compiler determines that the COBOL and PL/I mapping for the structure is different, and creation of a temporary structure mapped according to COBOL synchronized structure rules is not suppressed by one of the options NOMAP, NOMAPIN, and NOMAPOUT.

An entry for a FORTRAN mapped array, that is, an array passed to or from a FORTRAN program, has the word "FORTRAN" appended.

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

A separate entry will be made in the aggregate table for every aggregate dummy argument or FORTRAN mapped array 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 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.
- 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.

STATEMENT OFFSET ADDRESSES

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

Offsets given in 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 procedure or on-unit named in the message and then finding the largest entry in the table that is less than the offset in the message. If the procedure or on-unit name specified in the message is the same as that in the table (as it will be unless a secondary entry point is used), the statement will have been found.

If a secondary entry point is used the correct offset must be calculated.

The offset figure in the message is taken from the entry point used by the program and mentioned in the message. The offset used in the table is taken from the primary entry point of the procedure. If the entry points are not the same, the offset of the entry point must be added to the figure given in the execution time message and this figure used to establish the statement number.

In the program whose listing is shown below, the error message gives an offset of X'50' from the entry point A2. Entry point A2 is not the primary entry point. From the listing it can be seen that entry point A2 (statement 5) is at offset X'78'. To get the true offset, it is necessary to add the two figures and arrive at an offset of X'C8'. From the table it is clear that this offset is within statement 6.

```
          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;

TABLES OF OFFSETS AND STATEMENT NUMBERS
WITHIN PROCEDURE M

OFFSET (HEX)          0       56       5E
STATEMENT NO.         1       2        8

     WITHIN PROCEDURE A1

OFFSET (HEX)          0       78       A8       B4
STATEMENT NO.         3       5        4        6
```

Message:

```
IBM3011I 'ONCODE'=0320 'ZERODIVIDE'
   CONDITION RAISED AT OFFSET +000050 IN
   PROCEDURE WITH ENTRY A2
```

If a BEGIN block is involved, the offset to the BEGIN statement must be added before the process begins.

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 what are termed 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 in a communications area, the task communications area (TCA), used by the compiler and by the library subroutines for handling files and controlled variables.

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, commencing from 0001.

ADDR — 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 always starts with the standard entries shown in the table below, which assumes the existence of an external procedure called NAME.

External Symbol Dictionary

Symbol	Type	ID	Address	Length
PLISTART	SD	0001	000000	000050
***NAME1	SD	0002	000000	014538
***NAME2	SD	0003	000000	004F40
PLITABS	WX	0004	000000	
PLIXOPT	WX	0005	000000	
IBMBPOPT	WX	0006	000000	
PLIXHD	WX	0007	000000	
IBMBEATA	WX[1]	0008	000000	
PLIFLOW	WX	0009	000000	
PLICOUNT	WX	000A	000000	
IBMBPIRA	ER	000B	000000	
IBMBPIRB	ER	000C	000000	
IBMBPIRC	ER	000D	000000	
PLICALLA	LD		000006	
PLICALLB	LD		00000A	
PLIMAIN	SD	000E	000000	000008

[1] An ER type entry for IBMBEATA is produced if the INTERRUPT compiler option is specified.

PLISTART
SD-type entry for PLISTART. This control section transfers control to the initialization routine IBMBPIR. When initialization is complete, control passes to the address stored in the control section PLIMAIN. (Initialization is required only once during the execution of a PL/I program, 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 PLIMAIN.)

*****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.

IBMBPIRA
ER-type entry for IBMBPIRA, the entry point of the PL/I resident library subroutine that handles program initialization and termination.

OTHER ESD ENTRIES

The remaining entries in the external symbol dictionary vary, but generally include the following:

- SD-type entry for the 4-byte control section PLIMAIN, 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.
- Weak external reference to a number of housekeeping control sections as follows:

  PLITABS — A control section based on a structure that may be declared in the PL/I program to control formatting of stream files.

  PLIXOPT — Execution time options string control section.

  IBMBEATA — A module in the PL/I library used to set the attention exit for use in procedures compiled with the INTERRUPT option. This is an ER type entry if the procedure was compiled with the INTERRUPT option.

  PLIFLOW — A control section used to hold information generated by the FLOW option.

  PLICOUNT — A control section used to hold information generated by the COUNT option.

- 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. This list includes the names of resident library subroutines called directly by compiled code (first-level subroutines), and the names of other resident library subroutines that are called by the first-level subroutines.
- CM-type entries for nonstring element 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.

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

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 Assembler language.

Both a static internal storage map and the object listing contain information that cannot be fully understood without a knowledge of the structure of the object module. This is beyond the scope of this manual, but a full description of the object module, the static internal storage map, and the object listing can be found in OS PL/I Optimizing Compiler: Execution Logic.

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 informatory message that calls attention to a possible inefficiency in the program or gives other information generated by the compiler that may be of interest to you.

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 has applied a "fix-up" with confidence. The resulting program will execute and 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 execute but will not give correct results.

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

The compiler lists only those messages with a severity equal to or greater than that specified by the FLAG option, as shown in Figure 10 on page 55.

Each message is identified by an 8-character code of the form IELnnnnI, where:

- The first three characters "IEL" identify the message as coming from the optimizing compiler.
- The next four characters are a 4-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 messages publication for this compiler.

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

Figure 10. Selecting the Lowest Severity of Messages to be Printed, Using the FLAG Option

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 Figure 11.

Return Code	Meaning
0000	No error detected; compilation completed; successful execution anticipated.
0004	Possible error (warning) detected; compilation completed; successful execution probable.
0008	Error detected; compilation completed; successful execution probable.
0012	Severe error detected; compilation may have been completed; successful execution improbable.
0016	Unrecoverable error detected; compilation terminated abnormally; successful execution impossible.

Figure 11. Return Codes from Compilation of a PL/I Program

BATCHED COMPILATION

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 data set defined by the DD statement with the name SYSPUNCH, or in the data set defined by the DD statement with the name SYSLIN. 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 compiler options to be specified individually for each compilation. The first procedure may 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 may conflict with the requirements of subsequent procedures.

The method of coding a PROCESS statement and the options that may be included are described under "Specifying Compiler Options in the *PROCESS Statement" on page 13. 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. Furthermore, OBJECT, MDECK, and DECK may cause problems if they are specified on second or subsequent compilations but not on the first. This is because they require the opening of SYSLIN or SYSPUNCH and there may not be room for the associated data management routines and control blocks. When this happens compilation ends with 80A ABEND.

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, it is diagnosed and ignored. The compiler does not reorganize its storage between procedures of a batch.

NAME OPTION

The NAME option specifies that the compiler is to place a linkage editor NAME statement as the last 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 will be handled by the linkage editor. When the batch of object modules is link-edited, the linkage editor combines all the object modules between one NAME statement and the preceding NAME statement into a single load module; it takes the name of the load module from the NAME statement that follows the last object module that is to be included. When combining two object modules into one load module, the NAME option should not be used in the EXEC statement. An example of the use of the NAME option is given in Figure 12 on page 57.

```
//     EXEC PLIXC,PARM.PLI='LIST'
                    .
                    .
                    .
*  PROCESS NAME('A');
   ALPHA: PROC OPTIONS(MAIN);
                    .
                    .
                    .
          END ALPHA;
*  PROCESS;
   BETA: PROC;
                    .
                    .
                    .
          END BETA;
*  PROCESS NAME('B');
   GAMMA: PROC;
                    .
                    .
                    .
          END GAMMA;
```

Figure 12. Use of the NAME Option in Batched Compilation

Compilation of the PL/I procedures ALPHA, BETA, and GAMMA, would result in the following object modules and NAME statements:

```
  Object module for ALPHA
        NAME A (R)
  Object module for BETA
  Object module for GAMMA
        NAME B (R)
```

From this sequence of object modules and control statements, the linkage editor would produce two load modules, one named A containing the object module for the external PL/I procedure ALPHA, and the other named B containing the object modules for the external PL/I procedures BETA and GAMMA.

You should not specify the option NAME if you intend to process the object modules with the loader. The loader processes all object modules into a single load module; if there is more than one name, the loader recognizes the first one only and ignores the others.

RETURN CODES IN BATCHED COMPILATION

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

JOB CONTROL LANGUAGE FOR BATCHED PROCESSING

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

EXAMPLES OF BATCHED COMPILATIONS

If the external procedures are components of a large program and need to be executed together, they can be link-edited together and executed in subsequent job steps. Cataloged procedure PLIXCG can be used, as shown in Figure 13.

```
//OPT4#13 JOB
//STEP1 EXEC PLIXCG
//PLI.SYSIN DD *
            First PL/I source module
* PROCESS;
            Second PL/I source module
* PROCESS;
            Third PL/I source module
/*
//GO.SYSIN DD *
            Data processed by combined
            PL/I modules

/*
```

Figure 13. Example of Batched Compilation, Including Execution

If the external procedures are independent programs to be invoked individually from a load module library, cataloged procedure PLIXCL can be used. For example, a job that contains three compile-and-link-edit operations can be run as a single batched compilation, as shown in Figure 14.

```
//OPT4#14 JOB
//STEP1 EXEC PLIXCL,
// PARM.PLI='NAME(''PROG1'')',
// PARM.LKED=LIST
//PLI.SYSIN DD *
            First PL/I source program
* PROCESS NAME('PROG2');
            Second PL/I source program
* PROCESS NAME('PROG3');
            Third PL/I source program
/*
//LKED.SYSLMOD DD DSN=PUBPGM,
// DISP=OLD
```

Figure 14. Example of Batched Compilation, Excluding Execution

One of these programs, such as PROG2, can be invoked from the load module library as follows:

```
//OPTEX JOB
//JOBLIB DD DSNAME=PUBPGM,DISP=SHR
//J2 EXEC PGM=PROG2
//SYSIN DD *
            Data processed by program PROG2
/*
```

COMPILE-TIME PROCESSING (PREPROCESSING)

The preprocessing facilities of the compiler are described in the language reference manual for this compiler. You can include in a PL/I program statements 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 supplements the information contained in the language reference manual by providing some illustrations of the use of the preprocessor and explaining how to establish and use source statement libraries.

INVOKING THE PREPROCESSOR

The preprocessor stage of the compiler is executed if you specify the compiler option MACRO. The compiler and the preprocessor use the data set defined by the DD statement with the name SYSUT1 during processing. They also use this data set to store the preprocessed source program until compilation begins. The IBM-supplied cataloged procedures for compilation all include a DD statement with the name SYSUT1.

The term MACRO owes its origin to the similarity of some applications of the preprocessor to the macro language available with such processors as the IBM OS/VS-DOS/VSE-VM/370 Assembler. Such a macro language allows you to write a single instruction in a program to represent a sequence of instructions that have previously been defined.

The format of the preprocessor output is given in Figure 15.

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 less than 71 columns, then extra blanks are added on the right.
Columns 73-80	Sequence number, right-aligned. If either SEQUENCE or NUMBER apply, 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 has occurred while replacement is being attempted. If no error has occurred, the column is blank.

Figure 15. Format of the Preprocessor Output

Three other compiler options, MDECK, INSOURCE, and SYNTAX, are meaningful only when you also specify the MACRO option. All are described in detail under "Compiler Options" on page 11.

A simple example of the use of the preprocessor to produce a source deck for a procedure SUBFUN is shown in Figure 16 on page 60; according to the value assigned to the preprocessor variable USE, the source statements will represent either a subroutine or a function. The DSNAME used for SYSPUNCH specifies a source program library on which the preprocessor output will be placed. Normally compilation would continue and the preprocessor output would be compiled.

The language reference manual for this compiler describes how to use the %INCLUDE statement to incorporate source text from a library into a PL/I program. (A library is a partitioned data set that can be used for the storage of other data sets, termed members.) Source text that you may wish to insert into a PL/I program by means of a %INCLUDE statement must exist as a member within a library. Defining a source statement library to the compiler is described further under "Source Statement Library (SYSLIB)" on page 10.

```
//STEP1     EXEC  PLICLG
//PLI.SYSIN DD *
 MAKEIN:   PROC OPTIONS(MAIN);
     DCL IN FILE RECORD;
     DCL 1 CARD1,
           2 NAME     CHAR(10),
           2 NUMBER   CHAR(7),
           2 GARBAGE  CHAR(63);
     DCL 1 CARD2,
           2 NAME     CHAR(10),
           2 NUMBER   FIXED DEC(7),
           2 GARBAGE  CHAR(66);

         ON ENDFILE (SYSIN) GO TO PRINT;
         OPEN FILE(IN) OUTPUT;
 NEXT:   READ FILE(SYSIN) INTO(CARD1);
         CARD2 = CARD1, BY NAME;
         WRITE FILE(IN) FROM(CARD2);
         GO TO NEXT;

 PRINT: CLOSE FILE(IN);
        PUT FILE(SYSPRINT) PAGE;
        OPEN FILE(IN) SEQUENTIAL INPUT;
        ON ENDFILE(IN) GO TO FINISH;

 PRINTIN: READ FILE(IN) INTO(CARD2);
          PUT FILE(SYSPRINT) SKIP EDIT (CARD2) (A);
          GO TO PRINTIN;

 FINISH:  CLOSE FILE(IN);
 END MAKEIN;
/*
//GO.IN     DD DSN=HPU8.NEWLIB(IN),DISP=(NEW,KEEP),UNIT=SYSDA,
//     SPACE=(TRK,(1,1,1)),DCB=(RECFM=FB,LRECL=80,BLKSIZE=400)
//GO.SYSIN DD *
LOS ANGLES1234567
BUFFLO    0000000
PORTLAND  0000036
SAN FRAN  0001234
ST PAUL   9873640
SACRAMENTO0069872
COLUMBUS  0000000
DENVER     567000
SEATTLE         34
ROME      1234590
/*
```

Figure 16 (Part 1 of 2). Using the Preprocessor to Produce a Source Deck That Is Placed on a Source Program Library

```
//OPT4#8  JOB
//STEP2 EXEC  PLIXC,PARM.PLI='MACRO,MDECK'
//PLI.SYSPUNCH  DD  DSNAME=HPU8.NEWLIB(FUN),DISP=(NEW,KEEP),UNIT=SYSDA,
//          SPACE=(TRK,(1,1,1)),DCB=(RECFM=FB,LRECL=80,BLKSIZE=400)
//PLI.SYSIN  DD *
  SUBFUN: PROC(CITY)
          RETURNS(FIXED DEC(7));
     DCL IN FILE RECORD,
         1 DATA,
            2 NAME CHAR(10),
            2 POP FIXED DEC(7),
            2 GARBAGE CHAR(66);
     DCL CITY CHAR(10);
     %DCL USE CHAR;
     %USE='FUN';  /* FOR SUBROUTINE, SUBSTITUTE %USE='SUB' */

     OPEN FILE(IN);
     NAME='          ';

  NEXT: READ FILE(IN) INTO(DATA);
        PUT FILE(SYSPRINT) SKIP EDIT (DATA) (A);
        IF NAME=CITY THEN DO;
           CLOSE FILE(IN);
           %IF USE='FUN' %THEN %GOTO L1;
           PUT FILE(SYSPRINT) SKIP LIST(DATA);
        END;
        %GO TO L2;
  %L1:;
           RETURN(POP);
        END;
  %L2:;
        ELSE
           GO TO NEXT;
  END SUBFUN;
```

Figure 16 (Part 2 of 2). Using the Preprocessor to Produce a Source Deck That Is Placed on a Source Program Library

```
//OPT4#9  JOB
//STEP3  EXEC  PLIXC,PARM.PLI='M,INC,IS'
//PLI.SYSLIB DD  DSNAME=HPU8.NEWLIB(FUN),DISP=(OLD,KEEP),UNIT=SYSDA,
//     VOL=SER=nnnnnn,DCB=(RECFM=FB,LRECL=80,BLKSIZE=400)
//PLI.SYSIN  DD  *
  TEST: PROC OPTIONS(MAIN);
        DCL NAME CHAR(10),
            NO FIXED DEC(7);
        ON ENDFILE(SYSIN) GO TO FINISH;
  AGAIN: GET FILE(SYSIN) EDIT(NAME) (COLUMN(1),A(10));
        NO=SUBFUN(NAME);
        PUT FILE(SYSPRINT) SKIP EDIT('FOUND',NAME,NO)
           (A(6),A(10),F(7));
        GO TO AGAIN;
      %INCLUDE FUN;
   FINISH: END TEST;
/*
//GO.IN     DD DSN=HPU8.NEWLIB(IN),DISP=(OLD,KEEP),UNIT=SYSDA,
//     VOL=SER=nnnnnn,DCB=(RECFM=FB,LRECL=80,BLKSIZE=400)
//GO.SYSIN  DD  *
LOS ANGLES
SACRAMENTO
COLUMBUS
/*
```

Figure 17. Including Source Statements from a Library

The %INCLUDE statement may include one or more pairs of identifiers. Each pair of identifiers specifies the name of a DD statement that defines a library and, in parentheses, the name of a member of the library. For example, the statement:

```
%INCLUDE DD1 (INVERT),DD2(LOOPX);
```

specifies that the source statements in member INVERT of the library defined by the DD statement with the name DD1, and those in member LOOPX of the library defined by the DD statement with the name DD2, are to be inserted consecutively into the source program. The compilation job step must include appropriate DD statements.

If you omit the ddname from any pair of identifiers in a %INCLUDE statement, the ddname SYSLIB will be assumed. In such a case, you must include a DD statement with the name SYSLIB. (The IBM-supplied cataloged procedures do not include a DD statement with this name in the compilation procedure step.)

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

The use of a %INCLUDE statement to include the source statements for SUBFUN in the procedure TEST is shown in Figure 17 on page 61. The library HPU8.NEWLIB is defined in the DD statement with the qualified name PLI.SYSLIB, which is added to the statements of the cataloged procedure PLIXCL for this job. Since the source statement library is defined by a DD statement with the name SYSLIB, the %INCLUDE statement need not include a ddname.

It is not necessary to invoke the preprocessor if your source program, and any text to be included, contains no preprocessor statements other than %INCLUDE. Under these circumstances, faster inclusion of text can be obtained by specifying the INCLUDE compiler option.

DYNAMIC INVOCATION OF THE COMPILER

You can invoke the optimizing compiler from an Assembler language program by using one of the macro instructions ATTACH, CALL, LINK, or XCTL. The following information supplements the description of these macro instructions given in the supervisor and data management manual.

To invoke the compiler specify IELOAA as the entry point name.

You can pass three address parameters to the compiler:

1. The address of a compiler option list.
2. The address of a list of ddnames for the data sets used by the compiler.
3. The address of a page number that is to be used for the first page of the compiler listing on SYSPRINT.

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.

Note: If you want to pass parameters in an XCTL macro instruction, you must use the execute (E) form of the macro instruction. Remember also that the XCTL macro instruction indicates to the control program that the load module containing the XCTL macro instruction is completed. Thus the parameters must be established in a portion of main storage outside the load module containing the XCTL macro instruction, in case the load module is deleted before the compiler can use the parameters.

The format of the three parameters for all the macro instructions 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 compiler option keywords, separated by one or more blanks, a comma, or both of these.

DDNAME LIST

The ddname 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). Each entry in the list must occupy an 8-byte field; the sequence of entries is given in Figure 18.

Entry	Standard ddname
1	SYSLIN
2	not applicable
3	not applicable
4	SYSLIB
5	SYSIN
6	SYSPRINT
7	SYSPUNCH
8	SYSUT1
9	not applicable
10	not applicable
11	not applicable
12	not applicable
13	not applicable
14	SYSCIN

Figure 18. The Sequence of Entries in the DDname List

If a ddname is shorter than 8 bytes, fill the field with blanks on the right. If you omit an entry, fill its field with binary zeros; however, you may omit entries at the end of the list entirely.

PAGE NUMBER

The page number is contained in a 6-byte field beginning on a halfword boundary. The first halfword must contain the binary value 4 (the length of the remainder of the field). The last four bytes contain the page number in binary form.

The compiler will add 1 to the last page number used in the compiler listing and put this value in the page-number field before returning control to the invoking routine. Thus, if the compiler is reinvoked, page numbering will be continuous.

USING FAST PATH INITIALIZATION/TERMINATION (PL/I RELEASE 4)

The fast path initialization/termination feature reduces the time taken for initialization and termination of a PL/I program at the expense of a slight additional storage overhead in the load module. It is intended for installations where a large number of small programs are in use, such as IMS or other transaction oriented systems. (It is not intended for use with CICS where initialization and termination is handled in a different way; see Chapter 15, "Using PL/I on CICS" on page 360 for information on CICS.)

The feature can be installed during installation of the compiler and libraries. If it is installed, it applies to all programs. Your system programmer will tell you whether fast path initialization/termination is installed in your installation.

If the feature is installed, the following points should be borne in mind to make the best use of it:

- Execution time options should either be specified in the PLIXOPT string or be taken from installation defaults. Do not pass them as parameters.
- ISASIZE should be specified as a positive figure large enough to hold at least the initial storage requirements of the first block.
- NOSTAE and NOSPIE should be specified. (Using NOSTAE gives greater savings than using NOSPIE but both should be used where possible.)
- ON FINISH on-units should not be used.

An example of a suitable PLIXOPT string might be:

```
DCL PLIXOPT CHAR(50) VAR EXTERNAL INIT
  ('NOSTAE NOSPIE ISASIZE(5000)');
```

The REPORT option will help you to determine the ISA size that you need. To determine the minimum figure you should specify an ISASIZE of 1 and run with the REPORT option. The figure given in report output for "Length of Initial Storage Area" will then give you the minimum requirements for fast initialization. This will not, however, necessarily give the optimum performance; (see "Execution-Time Options" on page 31 for a full discussion on ISASIZE.)

In most installations using fast path initialization/termination, ISA size will not be critical as the PL/I programs will be small, and there will probably be enough space to make the installation default ISA size large enough to handle most programs.

CHAPTER 3. THE LINKAGE EDITOR AND THE LOADER

This chapter describes two processing programs of the operating system, the linkage editor and the loader. It explains the basic differences between them, describes the processing done by them, the JCL required to invoke them and, for the linkage editor, the additional processing it can do. Both processing programs are fully described in linkage editor and loader manuals.

The object module produced by the compiler from a PL/I program always requires further processing before it can be executed. This further processing, the resolution of external references inserted by the compiler, is performed either by the linkage editor or by the loader, both of which convert an object module into an executable program, which in the case of the linkage editor, is termed a load module.

The linkage editor and the loader require the same type of input, perform the same basic processing, and produce a similar type of output. The basic differences between the two programs lie in the subsequent form and handling of this output.

BASIC DIFFERENCES

The linkage editor converts an object module into a load module, and stores it in a program library in auxiliary storage. The load module becomes a permanent member of that library and can be retrieved at any time for execution in either the job that created it, or in any other job.

The loader, on the other hand, processes the object module, loads the processed output directly into main storage, and executes it immediately. The loader is essentially a one-shot program checkout facility; once the load module has been executed, it cannot be used again without reinvoking the loader. To keep a load module for later execution, or to provide an overlay structure, you must use the linkage editor.

When using the linkage editor, three job steps are required—compilation, link editing, and execution. When using the loader, only two job steps are required—compilation and execution.

CHOICE OF PROGRAM

If your installation includes both programs, the choice of program will depend on whether or not you want to retain a permanent copy of the load module, and on whether you want to use one of the facilities provided only by the linkage editor. All object modules acceptable to the linkage editor are acceptable to the loader; all load modules produced by the linkage editor, except those produced with the NE (not editable) attribute,[1] are also acceptable to the loader. The differences between the two programs are summarized below.

---

[1] The NE attribute is given to a load module that has no external symbol dictionary (ESD); a load module without an ESD cannot be processed again, either by the linkage editor or by the loader.

LINKAGE EDITOR

- The linkage editor converts an object module into a load module and stores it in a partitioned data set (program library) in auxiliary storage.
- The linkage editor can produce one or more load modules in a single step (for example, output from batch compilation).
- The linkage editor can accept input from other sources as well as from its primary input source and from the call library (SYSLIB).
- The linkage editor can provide an overlay structure for a program.

LOADER

- The loader converts an object module into an executable program in main storage and executes it immediately.
- The loader can produce only one load module in a single job step no matter how many object modules are produced (for example, the output from a batched compilation).
- The loader can accept input from its primary input source and from the call library (SYSLIB).

PERFORMANCE CONSIDERATIONS

If you use the loader, you will gain the advantage of a considerable saving in both time and auxiliary storage when running your PL/I program. Although the execution time will be unchanged, both the scheduling time and the processing time will be reduced, and much less auxiliary storage will be needed. These savings are achieved as follows:

Scheduling Time: Scheduling time for the loader is much less than that for link editing and execution because the loader needs only one job step.

Processing Time: The time taken to process an object module by the loader is approximately half that taken by the linkage editor to process the same module. This is achieved by the elimination of certain input/output operations required by the linkage editor, and by a reduction in module access time by the use of chained scheduling and improved buffering in the loader program.

Auxiliary Storage: The amount of auxiliary storage required by the loader when your job is compiled, loaded, and executed as a single job step, is much less than that required by the linkage editor because two of the standard data sets used by the linkage editor are not needed. If the loader input is to consist of existing load modules the auxiliary storage required for these can be reduced by storing them with unresolved external references. These external references are resolved by the loader.

MODULE STRUCTURE

Object and load modules have very similar structures; they differ only in that a load module that has been processed by the linkage editor contains certain descriptive information required by the operating system; in particular, the module is marked as "executable" or "not executable." A module comprises the following information:

- Text (TXT)
- External symbol dictionary (ESD)

- Relocation dictionary (RLD)
- END instruction

TEXT

The text of an object or load module consists of the machine instructions that represent the PL/I statements of the source program. These instructions are grouped together in what are termed control sections; a control section is the smallest group of machine instructions that can be processed by the linkage editor. An object module produced by the optimizing compiler includes the following control sections:

- Program control section: contains the executable instructions of the object module.
- Static internal control section: contains storage for all variables declared STATIC INTERNAL and for constants and static system blocks.
- Control sections termed common areas: one common area is created for each EXTERNAL file name or for each non-string element variable declared STATIC EXTERNAL without the INITIAL attribute.
- PLISTART: execution of a PL/I program always starts with this control section, which passes control to the appropriate initialization subroutine; when initialization is complete, control passes to the address stored in the control section PLIMAIN.
- Control sections for all PL/I library subroutines to be included with the program.

EXTERNAL SYMBOL DICTIONARY

The external symbol dictionary (ESD) is a table containing all the external symbols that appear in the object module. 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 names of the control sections are themselves external symbols, as are the names of variables declared with the EXTERNAL attribute and entry names in the external procedure of a PL/I program. References to external symbols defined elsewhere are also considered to be external symbols; they are known as external references. Such external references in an object module always include the names of the subroutines from either the OS PL/I Resident Library or the OS PL/I Transient Library that will be required for execution. They may also include calls to your own subroutines that are not part of the PL/I subroutine library, nor already included within the object module. The linkage editor or loader locates all the subroutines referred to, and includes them in the load module, or executable program respectively.

RELOCATION DICTIONARY

At execution time, the machine instructions in a load module use the following two methods of addressing locations in main storage:

1. Names used only within a control section have addresses relative to the starting point of the control section.
2. Other names (external names) have absolute addresses so that any control section can refer to them.

The relocation dictionary (RLD) contains information that enables absolute addresses to be assigned to locations within the load module when it is loaded into main storage for execution. These addresses cannot be determined earlier because the starting address is not known until the module is loaded. The relocation dictionaries from all the input modules are combined into a single relocation dictionary when a load module is produced.

END INSTRUCTION

This specifies the compiler-generated control section PLISTART as the entry point for the object module. It also contains "CSECT IDR" information for processing by the linkage editor. The CSECT IDR information is given in Figure 19.

Column	Information
33	The number of IDR entries that follow. This is always "1" for the optimizing compiler.
34 to 41	The program number of the compiler. (5734-PL1 for the optimizing compiler.)
44 to 47	The release number of the compiler. For example, '0102' indicates Release 1.2.
48 to 52	The date in year-day form (that is, yyddd).

Figure 19. The CSECT IDR Information

LINKAGE EDITOR

The linkage editor is an operating system processing program that produces load modules. It always stores the load modules in a library, from which the job scheduler can call them for execution.

The input to the linkage editor can include object modules, load modules, and control statements that specify how the input is to be processed. The output from the linkage editor comprises one or more load modules.

In addition to its primary function of converting object modules into load modules, the linkage editor can also be used to:

- Combine previously link-edited load modules.
- Modify existing load modules.
- Construct an overlay structure.

A load module constructed as an overlay structure can be executed in an area of main storage that is not large enough to contain the entire module at one time. The linkage editor divides the load module into segments that can be loaded and executed in turn.

LINKAGE EDITOR PROCESSING

A PL/I program, compiled by the optimizing compiler, cannot be executed until the appropriate library subroutines have been included. These subroutines are included in two ways:

1. By inclusion in the load module during link editing.

2. By dynamic call during execution.

The first method is used for most of the PL/I resident library subroutines; the following paragraphs describe how the linkage editor locates them. The second is used for the PL/I transient library subroutines, for example those concerned with input and output (including those used for opening and closing files), and those that generate execution-time messages.

In basic processing, as shown in Figure 20, the linkage editor accepts from its primary input source a data set defined by the DD statement with the name SYSLIN. For a PL/I program, this input is the object module produced by the compiler. The linkage editor uses the external symbol dictionary in this object module to determine whether the module includes any external references for which there are no corresponding external symbols in the module: it attempts to resolve such references by a method termed automatic library call.

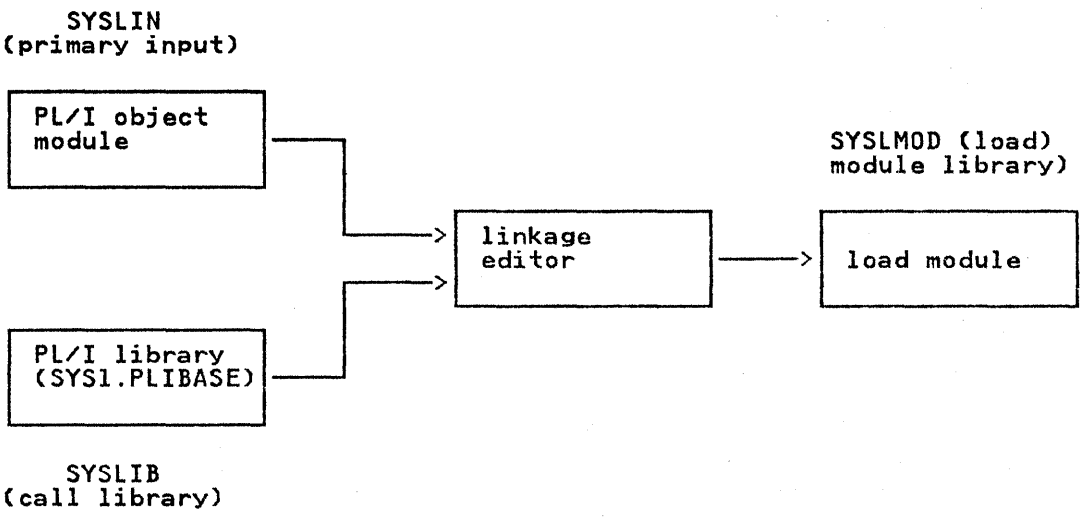


Figure 20. Basic Linkage Editor Processing

External symbol resolution by automatic library call involves a search of the data set defined by the DD statement with the name SYSLIB; for a PL/I program, this will be the PL/I resident library. The linkage editor locates the subroutines in which the external symbols are defined (if such subroutines exist), and includes them in the load module.

The linkage editor always places its output (that is, the load module) in the data set defined by the DD statement with the name SYSLMOD.

Any linkage editor processing additional to the basic processing described above must be specified by linkage editor control statements placed in the primary input. These control statement are described under "Additional Processing" on page 79.

JOB CONTROL LANGUAGE FOR THE LINKAGE EDITOR

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

The IBM-supplied cataloged procedures that include a link-edit procedure step are:

PLIXCL Compile and link edit

PLIXCLG Compile, link edit, and execute

PLIXLG Link edit and execute

The following paragraphs describe the essential JCL statements for link editing. The IBM-supplied cataloged procedures are described in Chapter 9, "Cataloged Procedures" on page 273, and include examples of these statements.

EXEC STATEMENT

The name of the linkage editor is HEWL.

The aliases IEWL or LINKEDIT are often used for the linkage editor.

The basic EXEC statement is:

```
//stepname EXEC PGM=IEWL
```

By using the PARM parameter of the EXEC statement, you can select one or more of the optional facilities provided by the linkage editor; these facilities are described under "Optional Facilities" on page 74.

DD STATEMENTS FOR THE STANDARD DATA SETS

The linkage editor always requires four standard data sets. You must define these data sets in DD statements with the ddnames SYSLIN, SYSLMOD, SYSUT1, and SYSPRINT.

A fifth data set, defined by a DD statement with the name SYSLIB, is necessary if you want to use automatic library call. The five data set names, together with other characteristics of the data sets, are shown in Figure 21.

ddname	Contents	Possible device classes[1]
SYSLIN	Primary input data, normally the compiler output	UNIT=SYSSQ or input job stream (specified by DD *)
SYSLMOD	Load module	UNIT=SYSDA
SYSUT1	Temporary workspace	UNIT=SYSDA
SYSPRINT	Listing, including messages	UNIT=SYSSQ (or SYSOUT=)

Figure 21 (Part 1 of 2). Linkage Editor Standard Data Sets

ddname	Contents	Possible device classes[1]
SYSLIB	Automatic call library (normally the PL/I resident library)	UNIT=SYSDA

Figure 21 (Part 2 of 2). Linkage Editor Standard Data Sets

[1] SYSSQ Magnetic tape or direct-access device
SYSDA Direct access device

PRIMARY INPUT (SYSLIN)

Primary input to the linkage editor must be a standard data set defined by a DD statement with the name SYSLIN; this data set must have consecutive organization. The input must comprise one or more object modules and/or linkage editor control statements; a load module cannot be part of the primary input, although it can be introduced by the control statement INCLUDE. For a PL/I program the primary input is usually a data set containing an object module produced by the compiler. This data set may be on magnetic tape or on a direct-access device, or you can include it in the input job stream. In all cases, the input must be in the form of 80-byte F-format records.

The IBM-supplied cataloged procedure PLIXLG includes the DD statement:

```
//SYSLIN DD DDNAME=SYSIN
```

This statement specifies that the primary input data set may be defined in a DD statement with the name SYSIN. If you use this cataloged procedure, specify this DD statement by using the qualified ddname LKED.SYSIN. For example, to link edit and execute an object module placed in the input stream, you can use the following statements:

```
//LEGO        JOB
//STEP1       EXEC PLIXLG
//LKED.SYSIN  DD *
      .
      .
      (insert here the object module to be
      link edited and executed)
      .
      .
/*
```

If object modules with identically named control sections appear in the primary input, the linkage editor processes only the first appearance of that control section.

You can include load modules or object modules from one or more libraries in the primary input by using a linkage editor INCLUDE statement as described under "Additional Processing" on page 79.

OUTPUT (SYSLMOD)

Output (that is, one or more load modules) from the linkage editor is always stored in a data set defined by the DD statement with the name SYSLMOD, unless you specify otherwise. This data set is usually called a library; libraries are fully described in Chapter 8, "Libraries of Data Sets" on page 264.

The IBM-supplied cataloged procedures include the following DD statement:

```
//SYSLMOD DD DSNAME=&&GOSET(GO),
//    UNIT=SYSDA,
//    DISP=(MOD,PASS),
//    SPACE=(1024,(50,20,1))
```

This statement defines a temporary library named &&GOSET and assigns the member name GO to the load module produced by the linkage editor. To retain the load module after execution of the job, replace this DD statement with one that defines a permanent library. For example, assume that you have a permanent library called USLIB on 3330 disk pack serial number 371; to name the load module MOD1 and place it in this library, code:

```
//LKED.SYSLMOD DD DSNAME=USLIB(MOD1),
//    UNIT=3330,VOL=SER=371,DISP=OLD
```

The SPACE parameter in the DD statement with the name SYSLMOD used in the IBM-supplied cataloged procedures allows for an initial allocation of 50K bytes and, if necessary, 15 further allocations of 20K bytes (a total of 350K bytes); this should suffice for most applications.

TEMPORARY WORKSPACE (SYSUT1)

The linkage editor requires a data set for use as temporary workspace. It is defined by a DD statement with the name SYSUT1. This data set must be on a direct-access device. The following statement contains the essential parameters:

```
//SYSUT1 DD UNIT=SYSDA,
// SPACE=(1024,(200,20))
```

You should normally never need to alter the DD statement with the name SYSUT1 in an IBM-supplied cataloged procedure, except to increase the SPACE allocation when processing very large programs.

If your installation supports dedicated workfiles, these can be used to provide temporary workspace for the link-edit job step, as described under "Compile and Link-Edit (PLIXCL)" on page 278.

AUTOMATIC CALL LIBRARY (SYSLIB)

Unless you specify otherwise, the linkage editor will always attempt to resolve external references by automatic library call (see "Linkage Editor Processing" on page 69). To enable it to do this, you must define the data set or data sets to be searched in a DD statement with the name SYSLIB. (To define second and subsequent data sets, include additional, unnamed, DD statements immediately after the DD statement SYSLIB; the data sets so defined will be treated as a single continuous data set for the duration of the job step.)

For a PL/I program, the DD statement SYSLIB will normally define the PL/I resident library. The subroutines of the resident library are stored in two data sets, SYS1.PLIBASE (the base library) and SYS1.PLITASK (the multitasking library). The base library contains all the resident library subroutines required by a nonmultitasking program. The multitasking library contains subroutines that are peculiar to multitasking, together with multitasking variants of some of the base library subroutines.

For link editing a nonmultitasking program, specify only the base library in the SYSLIB DD statement. The following DD statement will usually suffice:

```
//SYSLIB DD DSN=SYS1.PLIBASE,DISP=SHR
```

For link editing a multitasking program, specify both the multitasking library and the base library. When attempting to resolve an external reference, the linkage editor will first search the multitasking library; if it cannot find the required subroutine, it will then search the base library. To ensure that the search is carried out in the correct sequence, the DD statements defining the two sections of the library must be in the correct sequence: multitasking library first, base library second. The following DD statements will usually suffice:

```
//SYSLIB DD DSNAME=SYS1.PLITASK,DISP=SHR
//       DD DSNAME=SYS1.PLIBASE,DISP=SHR
```

LISTING (SYSPRINT)

The linkage editor generates a listing that includes reference tables relating to the load modules that it produces and also, when necessary, messages. The information that can appear is described under "Listing Produced by the Linkage Editor" on page 75.

You must define the data set on which you wish the linkage editor to store its listing in a DD statement with the name SYSPRINT. This data set must have consecutive organization. Although the listing is usually printed, it can be stored on any magnetic-tape or direct-access device. For printed output, the following statement will suffice:

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

EXAMPLE OF LINKAGE EDITOR JCL

A typical sequence of job control statements for link editing an object module is shown in Figure 22. The DD statement SYSLIN indicates that the object module will follow immediately in the input stream; for example, it might be an object deck created by invoking the optimizing compiler with the DECK option, as described under "DECK Option" on page 19. The DD statement with the name SYSLMOD specifies that the linkage editor is to name the load module LKEX, and that it is to place it in a new library name MODLIB; the keyword NEW in the DISP parameter indicates to the operating system that this DD statement specifies the creation of a library.

```
//LINK     JOB
//STEP1    EXEC PGM=IEWL
//SYSLMOD  DD DSNAME=MODLIB(LKEX),UNIT=3330,VOL=SER=D186,
//            SPACE=(CYL,(10,10,1)),DISP=(NEW,KEEP)
//SYSUT1   DD UNIT=SYSDA,SPACE=(1024,(200,20))
//SYSPRINT DD SYSOUT=A
//SYSLIB   DD DSNAME=SYS1.PLIBASE,DISP=SHR
//SYSLIN   DD *
                .
                .
    (insert here the object module to be link-edited)
                .
                .
/*
```

Figure 22. Typical Job Control Statements for Link-Editing a PL/I Program

OPTIONAL FACILITIES

The linkage editor provides a number of optional facilities that are selected by including the appropriate keywords in the PARM parameter of the EXEC statement that invokes it. Some of the more commonly used options are:

```
LIST
MAP or XREF
LET or XCAL
NCAL
RENT or REUS
SIZE
```

Code PARM= followed by the list of options, separating the options with commas and enclosing the list within single quotation marks, for example:

```
//STEPA EXEC PGM=IEWL,PARM='LIST,MAP'
```

If you are using a cataloged procedure, you must include the PARM parameter in the EXEC statement that invokes the procedure and qualify the keyword PARM with the name of the procedure step that invokes the linkage editor, for example:

```
//STEPA EXEC PLIXCLG,PARM.LKED='LIST,XREF'
```

Some of the linkage editor options are described in the following sections, in alphabetic order. For more detailed descriptions of these and other options, see the OS/VS Linkage Editor and Loader publication.

LET OPTION

The LET option specifies that the linkage editor is to mark the load module as "executable" even if slight errors or abnormal conditions have been found during link editing provided these do not exceed severity 2.

LIST OPTION

The LIST option specifies that all linkage editor control statements processed should be listed in the data set defined by the DD statement with the name SYSPRINT.

MAP OPTION

The MAP option specifies that the linkage editor is to produce a map of the load module showing the relative locations and lengths of all control sections in the load module.

NCAL OPTION

The NCAL option specifies that no external references are to be resolved by library call. However, the load module is marked "executable" provided that there are no errors.

You can use the NCAL option to conserve auxiliary storage in private libraries, since, by preventing the resolution of external references during link editing, you can store load modules without the relevant library subroutines; the DD statement with the name SYSLIB is not required. Before executing these load modules, you must link edit them again to resolve the external references, but the load module created need exist only while it is being executed. You can use this technique to combine separately compiled PL/I procedures into a single load module.

As a result of specifying NCAL and thus preventing the resolution of external references, the warning message IEW0461, together with a return code of 0004, may appear in the linkage editor listing for the PL/I program.

RENT OPTION

The RENT option specifies that the module is reenterable and can be executed by more than one task at a time.

REUS OPTION

The REUS option specifies that the module is serially reusable and can be executed by only one task at a time.

SIZE OPTION

The SIZE option specifies the amount of main storage, in bytes, to be allocated to the linkage editor. The syntax of the SIZE option is:

```
SIZE=(m[,n])
```

where "m" is the amount of main storage in bytes or K bytes (where K=1024) to be allocated to the linkage editor; it must include "n" and it must be greater than "n."

and "n" which is optional, is the amount of main storage (in bytes or K bytes) to be allocated to the load module buffer.

If you specify SIZE incorrectly, or if you omit it, default values set at system generation are used. If you specify SIZE greater than the region or partition size, the maximum amount of main storage will be used.

XCAL OPTION

The XCAL option specifies that the linkage editor will mark the load module as "executable" even if slight errors or abnormal conditions, including improper branches between control sections, have been found during link editing. XCAL, which implies LET, applies only to an overlay structure.

XREF OPTION

The XREF option specifies that the linkage editor is to print a map of the load module and a cross-reference list of all the external references in each control section. XREF implies MAP.

LISTING PRODUCED BY THE LINKAGE EDITOR

The linkage editor generates a listing, most of which is optional, that contains information about the link-editing process and the load module that it produces. It places this listing in the data set defined by the DD statement with the name SYSPRINT (usually output to a printer). The following description of the listing refers to its appearance on a printed page.

The listing comprises a small amount of standard information that always appears, together with those items of optional information specified in the PARM parameter of the EXEC statement that invokes the linkage editor, or that are applied by default. The optional components of the listing and the corresponding linkage editor options are as shown in Figure 23.

Listings	Options Required
Control statements processed by the linkage editor	LIST
Map of the load module	MAP or XREF
Cross-reference table	XREF

Figure 23. Linkage Editor Listings and Associated Options

The first page of the listing is identified by the linkage editor version and level number followed by a list of the linkage editor options used.

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

An example of the listing produced for a typical PL/I program is given in Appendix D, "Sample Program" on page 406

DIAGNOSTIC MESSAGES AND CONTROL STATEMENTS

The linkage editor generates messages, describing errors or conditions, detected during link editing, that may lead to error. These messages are listed immediately after the heading information on page 1 of the linkage editor listing. They are listed again at the end of the linkage editor listing under "Diagnostic Message Directory" on page 77.

If you have specified the option LIST, the names of all control statements processed by the linkage editor are listed immediately preceding the messages, and are identified by the 7-character code IEW0000.

Each message is identified by a similar 7-character code of the form IEWnnnx, where:

- The first three characters "IEW" identify the message as coming from the linkage editor.
- The next three characters are a 3-digit message number.
- The last character "x" is a severity code. The possible severity codes and their meanings are given in Figure 24 on page 77.

Severity Code	Meaning
0	A condition that will not cause an error during execution. The load module is marked as "executable."
1	A condition that may cause an error during execution. The load module is marked as "executable."
2	An error that could make execution impossible. The load module is marked as "not executable" unless you have specified the option LET.
3	An error that will make execution impossible. The load module is marked as "not executable."
4	An error that makes recovery impossible. Linkage editor processing is terminated, and no output other than messages is produced.

Figure 24. Diagnostic Message Severity Codes

At the end of the listing, immediately preceding the "Diagnostic Message Directory," the linkage editor places a statement of the disposition of the load module. See also "Diagnostic Message Directory." The disposition statements, with one exception, are self-explanatory; the exception is:

```
****modulename DOES NOT EXIST BUT HAS
BEEN ADDED TO DATA SET
```

This appears when the NAME statement has been used to add a new module to the data set defined by the DD statement with the name SYSLMOD. The use of the NAME statement is described under "Module Name" on page 79. If you name a new module by including its name in the DSNAME parameter of the DD statement with the name SYSLMOD, the linkage editor assumes that you want to replace an existing module (even if the data set is new).

DIAGNOSTIC MESSAGE DIRECTORY

When processing of a load module has been completed, the linkage editor lists in full all the messages whose numbers appear in the preceding list. The text of each message, an explanation, and any recommended programmer response, are given in the linkage editor and loader publication.

MODULE MAP

The linkage editor listing includes a module map only if you specify the options MAP or XREF. The map lists all the control sections in the load module and all the entry point names in each control section. The control sections are listed in order of appearance in the load module; alongside each control section name is its address relative to the start of the load module (address 0) and its length in bytes. The entry points within the load module appear on the printed listing below and to the right of the control sections in which they are defined; each entry point name is accompanied by its address relative to the start of the load module.

Each control section that is included by automatic library call is indicated by an asterisk. For an overlay structure, the control sections are arranged by segment in the order in which they are specified.

After the control sections, the module map lists the pseudo-registers established by the compiler. Pseudo-registers are fields in a communications area, the task communications area (TCA), used by PL/I library subroutines and compiled code during execution of a PL/I program. The main storage occupied by the TCA is not allocated until the start of execution of a PL/I program; it does not form part of the load module. The addresses given in the list of pseudo-registers are relative to the start of the TCA.

At the end of the module map, the linkage editor supplies the following information:

- The total length of the pseudo-registers.
- The relative address of the instruction with which execution of the load module will commence (ENTRY ADDRESS).
- The total length of the load module. For an overlay structure, the length is that of the longest path.

All the addresses and lengths given in the module map and associated information are in hexadecimal.

CROSS-REFERENCE TABLE

The linkage editor listing includes a "Cross-Reference Table" only if you specify the option XREF. This option produces a listing that comprises all the information described under "Module Map" on page 77, together with a cross-reference table of external references. The table gives the location of each reference within the load module, the symbol to which the reference refers, and the name of the control section in which the symbol is defined.

For an overlay structure, a cross-reference table is provided for each segment. It includes the number of the segment in which each symbol is defined.

Unresolved symbols are identified in the cross-reference table by the entries $UNRESOLVED or $NEVER-CALL. An unresolved weak external reference (WXTRN) is identified by the entry $UNRESOLVED(W).

RETURN CODE

For every linkage editor job or job step, the linkage editor 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 and is derived by multiplying the highest severity code by four, as shown in Figure 25 on page 79. (See also "Diagnostic Message Directory" on page 77.)

Return Code	Meaning
0000	No messages issued; link editing completed without error; successful execution anticipated.
0004	Warning messages only issued; link editing completed; successful execution probable.
0008	Error messages only issued; link editing completed; execution may fail.
0012	Severe error messages issued; link editing may have been completed, but with errors; successful execution improbable.
0016	Unrecoverable error message issued; link editing terminated abnormally; successful execution impossible.

Figure 25. Return Codes from the Linkage Editor

ADDITIONAL PROCESSING

Basic processing by the linkage editor produces a single load module from the data that it reads from its primary input, but it has several other facilities that you can call upon by using linkage editor control statements. The use of those statements of particular relevance to a PL/I program is described below. All the linkage editor control statements are fully described in the linkage editor and loader publication.

FORMAT OF CONTROL STATEMENTS

A linkage editor control statement is an 80-byte record that contains two fields. The operation field specifies the operation required of the linkage editor; it must be preceded and followed by at least one blank character. The operand field names the control sections, data sets, or modules that are to be processed, and it may contain symbols to indicate the manner of processing; the field consists of one or more parameters separated by commas. Some control statements may have multiple operand fields separated by commas.

The position of a control statement in the linkage editor input depends on its function.

In the following descriptions of the control statements, items within brackets [] are optional.

MODULE NAME

A load module must have a name so that the linkage editor and the operating system can identify it. A name comprises up to eight characters, the first of which must be alphabetic.

You can name a load module in one of two ways:

1. If you are producing a single load module from a single link-edit job step, it is sufficient to include its name as a member name in the DSNAME parameter of the DD statement with the name SYSLMOD.

2. If you are producing two or more load modules from a single link-edit job step, you will need to use the NAME statement. (The optimizing compiler can supply the NAME statements when you use batch compilation as described in "NAME Option" on page 23).

The syntax of the NAME statement is:

```
NAME name[(R)]
```

where "name" is any name of up to eight characters; the first character must be alphabetic. The NAME statement serves the following functions:

- It identifies a load module. The name specified will be given to the load module. "(R)," if present, specifies that the load module is to replace an existing load module of the same name in the data set defined by the DD statement with the name SYSLMOD.
- It acts as a delimiter between input for different load modules in one link-edit step.

The NAME statement must appear in the primary input to the linkage editor (the standard data set defined by the DD statement SYSLIN); if it appears elsewhere, the linkage editor ignores it. The statement must follow immediately after the last object module that will form part of the load module it names (or after the INCLUDE control statement that specifies the last object module).

ALTERNATIVE NAMES

You can use the ALIAS statement to give a load module an alternative name; a load module can have as many as sixteen aliases in addition to the name given to it in a DD statement with the name SYSLMOD, or by a NAME statement.

The syntax of the ALIAS statement is:

```
ALIAS name
```

where "name" is any name of up to eight characters; the first character must be alphabetic. You can include more than one name in an ALIAS statement, separating the names by commas, for example:

```
ALIAS FEE,FIE,FOE,FUM
```

An ALIAS statement can be placed before, between, or after object modules and control statements that are being processed to form a load module, but it must precede the NAME statement that specifies the primary name of the load module.

To execute a load module, you can include an alias instead of the primary name in the PGM parameter of an EXEC statement.

Aliases can be used for external entry points in a PL/I procedure. Hence a CALL statement or a function reference to any of the external entry names will cause the linkage editor to include the module containing the alias entry names without the need to use the INCLUDE statement for this module.

ADDITIONAL INPUT SOURCES

The linkage editor can accept input from sources other than the primary input defined in the DD statement with the name SYSLIN. For example, automatic library call enables the linkage editor to include modules from a data set (a library) defined by the DD statement with the name SYSLIB. You can name these additional input sources by means of the INCLUDE statement, and you can direct the library call mechanism to alternative libraries by means of the LIBRARY statement.

INCLUDE STATEMENT

The INCLUDE statement causes the linkage editor to process the module or modules indicated. After the included modules have been processed, the linkage editor continues with the next item in the primary input. If an included sequential data set also contains an INCLUDE statement, that statement is processed as if it were the last item in the data set, as shown in Figure 26.

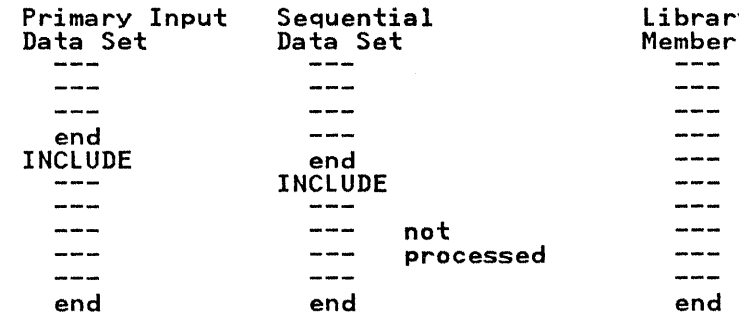


Figure 26. Processing Additional Data Sources

The syntax of the INCLUDE statement is:

```
INCLUDE ddname[(membername)]
```

where "ddname" is the name of a DD statement that defines either a sequential data set or a library that contains the modules and control statements to be processed. If the DD statement defines a library, replace "membername" with the names of the modules to be processed, separated by commas. You can specify more than one ddname, each of which may be followed by any number of member names in a single INCLUDE statement. For example:

```
INCLUDE D1(MEM1,MEM2),D2(MODA,MODB)
```

specifies the inclusion of the members MEM1 and MEM2 from the library defined by the DD statement with the name D1, and the members MODA and MODB from the library defined by the DD statement with the name D2.

LIBRARY STATEMENT

The basic function of the LIBRARY statement is to name call libraries in addition to those named in the DD statement SYSLIB. The syntax of the LIBRARY statement is:

```
LIBRARY ddname(membername)
```

where "ddname" is the name of a DD statement that defines the additional call library, and "membername" is the name of the module to be examined by the call mechanism. More than one module can be specified; separate the module names with commas.

OVERLAY STRUCTURES

A load module constructed as an overlay structure can be executed in an area of main storage that is not large enough to contain the entire module at one time. The linkage editor divides the load module into segments that can be loaded and executed in turn. To construct an overlay structure, you must use linkage editor control statements to specify the relationship between the segments. One segment, termed the root segment must remain in main storage throughout the execution of the program.

In an overlay environment the addressing of a static external structure element, array, or string may be incorrect if used in a data-directed I/O statement or CHECK statement. This error will arise if the control section containing the symbol table of the identifier, and the corresponding static internal control section are not in the same overlay segment. This is because the symbol table contains the address of a locator that is in static internal storage. The difficulty can be avoided by ensuring that the procedure in the root segment contains a reference to the identifier in a data-directed I/O or CHECK context. The statement containing the identifier need not be executed, but you must ensure that it is not removed by optimization; its presence ensures that the symbol table for the identifier addresses the locator in the static internal control section of the root segment.

The descriptor for a controlled external aggregate with fixed extents is stored in the static internal control section of the procedure that allocates it. This prevents references to the external variable being made in other procedures that overlay the segment in which it was allocated. A controlled external variable must be allocated in one of two ways:

1. The variables can be allocated in the root phase. A convenient technique to use would be to have a subroutine, containing the ALLOCATE statement, which could be called from any segment.

2. The variable can be allocated with adjustable extents, so that the descriptor will be copied into the controlled storage area when allocation takes place. Note that this method uses more storage.

DESIGN OF THE OVERLAY STRUCTURE

Before preparing the linkage editor control statements, you must design the overlay structure for your program. A tree is a graphic representation of an overlay structure that shows which segments occupy main storage at different times. The design of trees is discussed in the linkage editor and loader publication, but for the purposes of this chapter, Figure 27 on page 83 contains a simple example. The program comprises six procedures: A, B, C, D, E, and F. Procedure B calls procedure C which, in turn, calls procedures D and E. (Only procedure A requires the option MAIN.)

```
A: PROC OPTIONS(MAIN);
    .
    CALL B;
    .
    CALL F;
    .
    END A;

B: PROC;
    .
    CALL C;
    .
    END B;

C: PROC;
    .
    CALL D;
    .
    CALL E;
    .
    END C;

D: PROC;
    .
    .
    END D;

E: PROC;
    .
    .
    END E;

F: PROC;
    .
    .
    END F;
```

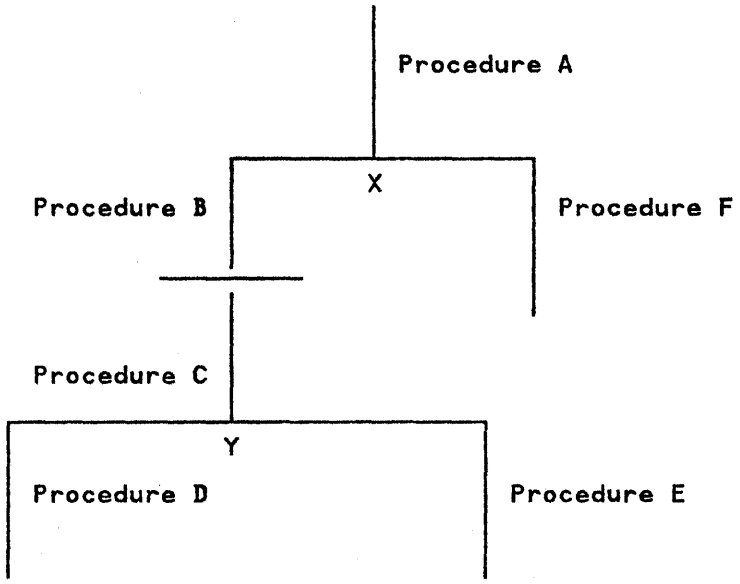


Figure 27. Overlay Structure and Its Tree

The main procedure (A) must be in main storage throughout the execution of the program. Since the execution of procedure B will be completed before procedure F is called, the two procedures can occupy the same storage; this is depicted by the lines representing the two procedures in Figure 26 on page 81 starting from the common point (node) X. Procedure B must remain in storage while procedures C, D, and E are executed, but procedures D and E can occupy the same storage; thus the lines representing procedures D and E start from node Y.

The degree of segmentation that can be achieved can be clearly seen from the figure. Since procedure A must always be present, it must be included in the root segment. Procedures F, D and E can usefully be placed in individual segments, as can procedures B and C be placed together; there is nothing to be gained by separating procedures B and C, since they must be present together at some time during execution.

CONTROL STATEMENTS

Two linkage editor control statements, OVERLAY and INSERT, control the relationship of the segments in the overlay structure. The OVERLAY statement specifies the start of a segment and the INSERT statement specifies the positions of control sections in a segment. You must include the attribute OVLY in the PARM parameter of the EXEC statement that invokes the linkage editor, otherwise the linkage editor will ignore the control statements.

The syntax of the OVERLAY statement is:

```
OVERLAY symbol
```

where "symbol" is the node at which the segment starts (for example, X in Figure 27 on page 83). You must specify the start of every segment, except the root segment, in an OVERLAY statement.

The syntax of the INSERT statement is:

```
INSERT control-section-name
```

where "control-section-name" is the name of the control section (that is, the derivative of the procedure name that is found in the linkage editor map) that is to be placed in the segment. More than one control section can be specified, separate the names with commas. The INSERT statements that name the control sections in the root segment must precede the first OVERLAY statement.

CREATING AN OVERLAY STRUCTURE

The most efficient method of defining an overlay structure, and the simplest for a PL/I program, is to group all the OVERLAY and INSERT statements together and place them in the linkage editor input (SYSLIN) after the object modules that form the program. The linkage editor initially places all these object modules in the root segment, and then moves those control sections that are referred to in INSERT statements into other segments.

This method has the advantage that you can use batched compilation to process all the procedures in one job step and place the object modules in a temporary data set; this data set must have consecutive organization. You can then place the linkage editor control statements in the input stream, concatenating them with the data set that contains the object modules. (Do not use the NAME compiler option to name the object modules; if you do, the NAME statements inserted by the compiler will cause the linkage editor to attempt to create separate load modules rather than a single overlay structure.)

The use of the IBM-supplied cataloged procedure PLIXCLG to create and execute the overlay structure of Figure 27 on page 83 is shown in Figure 28 on page 85.

```
//OPT5#12    JOB
//STEP1      EXEC PLIXCLG,
//           PARM.LKED='OVLY'
//PLI.SYSIN  DD *

  (insert here source statements for procedure A)

 * PROCESS;

  (insert here source statements for procedure B)

 * PROCESS;

  (insert here source statements for procedure C)

 * PROCESS;

  (insert here source statements for procedure D)

 * PROCESS;

  (insert here source statements for procedure E)

 * PROCESS;

  (insert here source statements for procedure F)

 /*
 //LKED.SYSIN DD *
    OVERLAY X
    INSERT ******B1,******C1
    OVERLAY Y
    INSERT ******D1
    OVERLAY Y
    INSERT ******E1
    OVERLAY X
    INSERT ******F1
 /*
```

Figure 28. Creating and Executing the Overlay Structure of Figure 27.

An alternative approach instead of batched compilation is to compile the procedures independently and store them as object modules in a private library. You can then use an INCLUDE statement to place them in the input to the linkage editor (SYSLIN).

If an INSERT statement contains the name of an external procedure, the linkage editor will move only the related program control section that has the same name. All other control sections established by the compiler, and all the PL/I library subroutines, will remain in the root segment.

It is important that the PL/I library subroutines be in the root segment, since the optimizing compiler does not support exclusive calls (calls between segments that do not lie in the same path). For example, in Figure 27 on page 83, procedures in the segment containing D could call procedures in the segments containing A, B, C, and D, but not in the segments containing E or F. Procedures in the segments containing B or C could call procedures in the segments containing A, B, C, D, and E, but not in the segment containing F. A procedure in the segment containing B may not call a procedure in the segment containing A if this latter procedure calls a procedure in the segment containing F.

However, certain library subroutines may not be required by all segments, in which case you can move them into a lower segment.

To do this, compile the procedures using the compiler option ESD, and examine the resulting external symbol dictionary. For example, if in Figure 27 on page 83, a library subroutine is called only by the segment containing E, you can move it into that segment by placing an INSERT statement, specifying the subroutine name, immediately after the statement INSERT ******E1.

Similarly, you can move control sections from the root segment to lower segments. For example, to move the static internal control section for procedure F into the segment containing F, place the statement INSERT ******F2 after the statement INSERT ******F1. Values assigned to static data items are not retained when a segment is overlaid. Do not move static data from the root segment unless it comprises only:

- Values set by the INITIAL attribute and then unchanged (that is, read-only data).
- Values that need not be retained between different loadings of the segment.

Care must be taken to ensure that the static external control sections for all the PL/I files used in an overlay program are placed in the root segment. If this is not done, failures may occur when the ERROR condition is raised and the PL/I error routines attempt to close the files. In particular, the static external control section for SYSPRINT must always be placed in the root segment.

When using the COUNT option, ensure that all procedures for which count information is required have their static internal control sections in the root segment, or the count information will be rendered invalid.

LINK EDITING MULTIPLE OBJECT MODULES.

When a PL/I MAIN procedure is link-edited with other object modules produced by the PL/I compiler, the entry point of the resulting load module will be resolved to the external symbol PLISTART. This will happen automatically, because the PLISTART CSECT is generated first in the PL/I object module output and is nominated in the END statement of the object module. Execution-time errors will occur if the load module entry point is forced to some other symbol by use of the linkage editor ENTRY control statement. See Chapter 5 of the Execution Logic Manual for details on the initialization of a PL/I MAIN procedure.

If a PL/I MAIN procedure is link-edited with object modules produced by other language processors or by the assembler and is the first module to receive control, the user must ensure that the entry point of the resulting load module is resolved to the external symbol PLISTART. This may be done most conveniently by ensuring that the PL/I object module is first in the input to the linkage editor. Alternatively, the following linkage editor ENTRY control statement may be included in the input to the linkage editor:

```
ENTRY PLISTART
```

If you want to pre-link PL/I subroutines, store them in a load library, and later "INCLUDE" them with main procedures, the subroutines must be linked with the NCAL linkage editor option. The NCAL option will cause unresolved external reference error messages from the linkage editor, but these will be resolved when the PL/I main procedure is linked with the subroutines. The NCAL option is needed because, in a PL/I load module, all the resident modules must be at the same level, and not resolving external references until the final link will ensure this.

Figure 29 shows an example of link-editing a PL/I object module with FORTRAN and COBOL object modules.

```
//JOBNAME JOB
//* ************************************************
//*   LINK-EDITING PL/I WITH FORTRAN AND COBOL   */
//*   PL1 INVOKES FORTRAN WHICH INVOKES COBOL    */
//* ************************************************
//PL1          EXEC PLIXC,
//             PARM.PLI='OBJECT'
//PLI.SYSLIN   DD DSN=&&LOADSET,SPACE=...
//PLI.SYSIN    DD DSN=STEP1.TEST.PL1...
//* ************************************************
//*   CALL A FORTRAN SUBPROGRAM STEP2.TEST.FORT     */
//* ************************************************
//FORT2        EXEC FTG1C
//FORT.SYSLIN DD DSN=&&LOADSET,...
//FORT.SYSIN   DD DSN=STEP2.TEST.FORT,...
//* ************************************************
//*   CALL A COBOL SUBPROGRAM STEP3.TEST.COBOL      */
//* ************************************************
//COBOL3       EXEC COBUC,PARM.COB='NODECK,LOAD,APOST'
//COB.SYSLIN   DD DSN=&&LOADSET,...
//COB.SYSIN    DD DSN=STEP3.TEST.COBOL,...
//* ************************
//*    LINK-EDIT STEP       */
//* ************************
//LKEDALL      EXEC PLIXLG
//LKED.SYSLIB DD ...
//             DD DSN=SYS1.FORTLIB,DISP=SHR
//             DD DSN=SYS1.COBLIB,DISP=SHR
//             DD DSN=SYS1.PPLINK,DISP=SHR
//LKED.SYSIN   DD DSN=&&LOADSET,DISP=(OLD,DELETE)
//GO.OUT       DD SYSOUT=A,...
//GO.SPACE     DD UNIT=SYSDA,...
//GO.SYSOUT    DD SYSOUT=A
```

Figure 29. Link-Editing PL/I with Other High Level Languages

LINK-EDITING FETCHABLE LOAD MODULES

The PL/I FETCH and RELEASE statements permit the dynamic loading of separate load modules which can be subsequently invoked from the PL/I object program. There are a number of restrictions on the PL/I statements that can be used in fetched procedures. These are described in the Language Reference Manual for this compiler.

Fetchable (or dynamically-loaded) modules should be link-edited into a load module library which is subsequently made available for the job step by means of a JOBLIB or STEPLIB DD statement.

The step which link-edits a fetchable load module into a library requires the following linkage editor control statements:

- An ENTRY statement to define the entry-point into the PL/I program.
- A NAME statement to define the name used for the fetchable load module. This statement is required if the compiler option NAME is not used and if the name is not specified in the DSN parameter in the SYSLMOD DD statement used to define the load module library.
- Optionally, for optimum space saving, REPLACE statements to delete the control sections shown in Figure 30 on page 88, if they are present in the object module.

Control Section	Present In
PLISTART	All programs
IBMBJWT1	Programs that use the WAIT statement
IBMTJWT1	Multitasking programs that use the WAIT statement
IBMBTOC1	Programs that use the COMPLETION built-in function or pseudovariable
IBMTTOC1	Multitasking programs that use the COMPLETION built-in function or pseudovariable
IBMBTPR1	Programs that use the PRIORITY pseudovariable
IBMTTPR1	Multitasking programs that use the PRIORITY pseudovariable
IBMBEFL1	Programs compiled with the FLOW or COUNT options

Figure 30. Control Sections to be Deleted for Optimum Space-Saving

The name or any alias by which the fetchable load module is identified in the load module library must appear in a FETCH or RELEASE statement within the scope of the invoking procedure.

COBOL or FORTRAN modules cannot be loaded dynamically by the PL/I FETCH statement.

The job control statements and the linkage editor statements to link-edit a fetchable load module into a library called PRVLIB are given in Figure 31. The cataloged procedure PLIXCL is used to illustrate these statements by sharing a job that includes both the compilation and the link-editing of the fetchable PL/I module.

```
//FETCH JOB
//STP EXEC PLIXCL
//PLI.SYSIN DD *
     .
     .
     .
     PL/I source(fetchable)
     .
     .
     .
/*
//LKED.SYSLMOD DD DSN=PRVLIB,...
//LKED.SYSLIN DD *
  ENTRY    procedure-name
  REPLACE PLISTART
  REPLACE IBMBJWT1
  REPLACE IBMBTOC1
  REPLACE IBMBTPR1
  INCLUDE OBJMOD
  NAME FETCH1
/*
//LKED.OBJMOD  DD  DSN=&&LOADSET,DISP=(OLD,...
```

Figure 31. Example of Link-Editing a Fetchable Load Module

MULTITASKING CONSIDERATIONS

When fetchable load modules are called as tasks, certain library routines are assumed to be available in the main task load module. Therefore, if a fetchable task uses one of them, and the main does not, then that library routine must be added to the main task load module via a link-edit INCLUDE statement.

The affected functions and their required library routines are:

Function	Library Routine
GOTO out of block	IBMBPGOA
WAIT	IBMBJWTA
EVENT pseudovariable	IBMBTOCA
COMPLETION pseudovariable	IBMBTOCA
PRIORITY pseudovariable	IBMBTPRA
EXCLUSIVE file attribute	IBMBPQDA

EXTENDED ARCHITECTURE CONSIDERATIONS

If only Release 5 object modules and resident library routines are link-edited together, then the resulting load module is RMODE(ANY) and AMODE(31); no linkage editor parameters or control statements are required for this result. This load module is compatible with 31-bit execution on MVS/Extended Architecture, and is also compatible with execution on MVS/SP 1.3.

If you require changes to the modes of the load module, then you specify parameters using the linkage editor JCL or using a linkage editor control statement.

JCL parameters are specified as

```
PARM='...[,RMODE={24|ANY},AMODE={24|31|ANY},]
```

and control statement as

```
MODE AMODE(24|31|ANY),RMODE(24|ANY)
```

For more information on OS PL/I and Extended Architecture, see Appendix F, "MVS/Extended Architecture (MVS/XA) Considerations" on page 453. For more information on the linkage editor, see MVS/Extended Architecture Linkage Editor and Loader User's Guide.

COMBINING PL/I MODULES FROM THE OPTIMIZING AND CHECKOUT COMPILERS

For information about combining PL/I modules from the optimizing and checkout compilers, see the OS PL/I Checkout Compiler: Programmer's Guide. OS PL/I Release 5 object code and Release 5 transient library routines will execute with the OS PL/I Checker Release 3.0. The Release 5 object modules must not use the resident library routines that are part of the Checker (PLICMIX).

LOADER

The loader is a program that produces and executes load modules. It always stores the load modules directly in main storage where they are executed immediately.

The input to the loader can include single object modules or load modules, several object modules or load modules, or a mixture of both. The output from the loader always comprises an executable program that is loaded into main storage from where it will be executed.

Unlike the linkage editor you cannot use any control statements with the loader. If any linkage editor control statements are used, they will be ignored, and their presence in the input stream will not be treated as an error. Your job will continue to be processed, a message will be generated and, if you have included a DD statement with the name SYSLOUT, this message and the name of the control statement will be printed on your listing.

The loader compensates for the absence of the facilities provided by control statements by allowing the concatenation of both object and load modules in the data set defined by the DD statement with the name SYSLIN, and by allowing an entry point to be specified by means of the EP option, as described under "Optional Facilities of the Loader" on page 96.

LOADER PROCESSING

A PL/I program cannot be executed until the appropriate PL/I library subroutines have been included. All library subroutines are included during loading. In basic processing, as shown in Figure 33 on page 91, the loader accepts data from its primary input source, a data set defined by the DD statement with the name SYSLIN. For a PL/I program, this data is the object module produced by the compiler. The loader uses the external symbol dictionary in this object module to determine whether the module includes any external references for which there are no corresponding external symbols in the module: it attempts to resolve such references by a method termed automatic library call as described in "Linkage Editor Processing" on page 69.

The loader locates the subroutines in which the external symbols are defined (if such subroutines exist) and includes them in the load module. If all external references are resolved satisfactorily the load module is executed.

The loader will always search the link-pack area before searching the PL/I resident library, as shown in Figure 34 on page 92. The link-pack area is an area of main storage in which frequently used load modules are stored permanently. If there is more than one copy of an object module in the data set defined by the DD statement with the name SYSLIN, the loader will load the first and ignore the rest.

MAIN STORAGE REQUIREMENTS

The minimum main storage requirements for the loader are shown in Figure 32.

Storage Required for:	Amount (min) in Bytes
Loader program	10K
Data management access routines	4K
Buffers and tables used by loader	3K
PL/I program to be executed	variable

Figure 32. Main Storage Requirements for the Loader

This amounts to at least 17K bytes for the loader and its associated routines and data areas plus the main storage required for the program that is to be executed. If the loader program and the data management access routines were stored in the link-pack area, the amount of main storage required would be 3K bytes for the loader data area plus that required by the program that is to be executed.

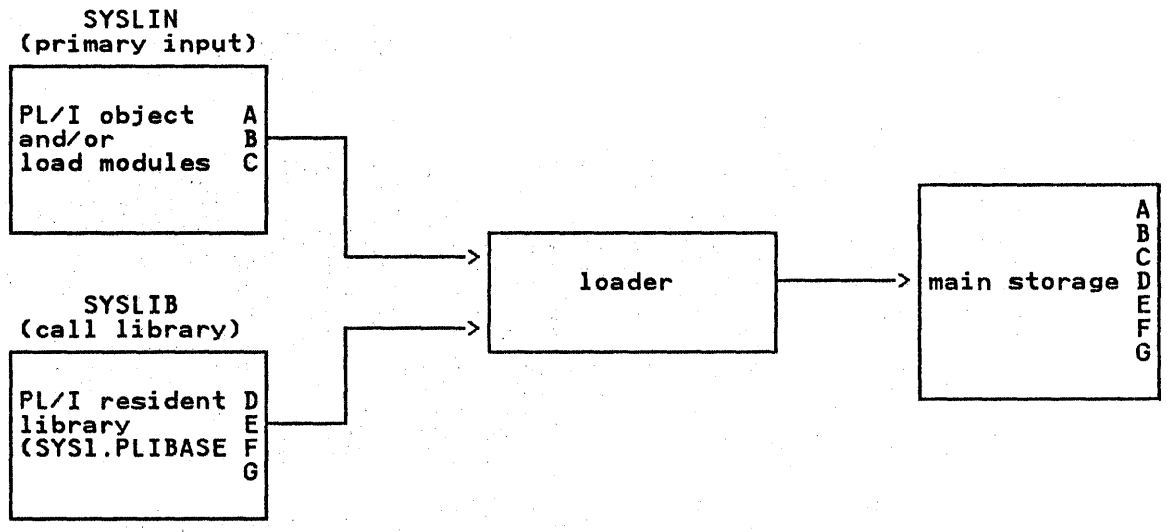


Figure 33. Basic Loader Processing

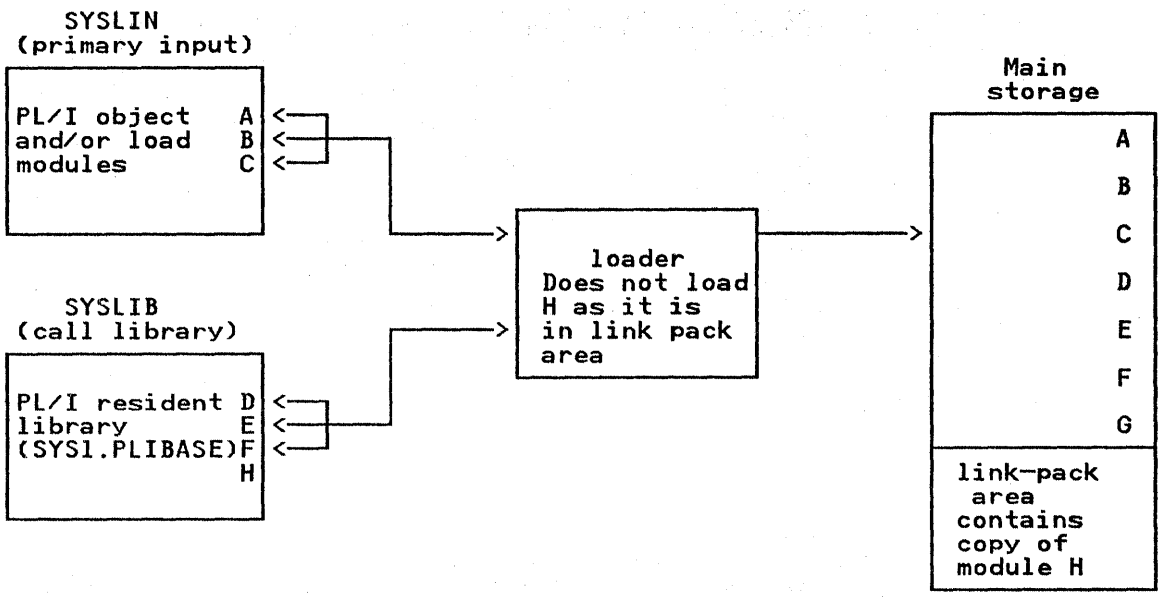


Figure 34. Loader Processing, Link-Pack Area and SYSLIB Resolution

ddname	Contents of Data Set	Possible Device Classes[1]
SYSLIN	Primary input (normally the output from the compiler)	SYSSQ or the input job stream (specified by DD *)
SYSLOUT	Loader messages and module map listing	SYSSQ, SYSDA, or SYSOUT=A
SYSPRINT	PL/I execution-time messages and problem output listing	SYSSQ, SYSDA, or SYSOUT=A
SYSLIB	Automatic call library	SYSDA

Figure 35. Loader Standard Data Sets

Note to Figure 35:

[1] SYSSQ Magnetic tape or direct-access device

SYSDA Direct-access device

SYSOUT=A Normal printed output class for system output

JOB CONTROL LANGUAGE FOR THE LOADER

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

The IBM-supplied cataloged procedures that include a loader procedure step are as follows:

PLIXCG Compile, load-and-execute

PLIXG Load-and-execute

The following paragraphs describe the essential JCL statements for the loader. The IBM-supplied cataloged procedures are described under Chapter 9, "Cataloged Procedures" on page 273, and include examples of these statements.

EXEC STATEMENT

The name of the loader is IEWLDRGO. It also has the alias LOADER, which is used in the IBM-supplied cataloged procedures, and will be used to refer to the loader program in the rest of this chapter. The basic EXEC statement is:

```
//stepname  EXEC PGM=LOADER
```

By using the PARM parameter of the EXEC statement, you can select one or more of the optional facilities provided by the loader; these are described under "Optional Facilities of the Loader" on page 96.

DD STATEMENTS FOR THE STANDARD DATA SETS

The loader always requires one standard data set; that defined by the DD statement with the name SYSLIN. Three other standard data sets are optional and if you use them you must define them in DD statements with the names SYSLOUT, SYSPRINT, and SYSLIB. The four data sets, their names, and other characteristics of the data sets, are shown in Figure 35 on page 92.

The data sets defined by the DD statements with the names SYSLIN, SYSLIB, and SYSLOUT are those specified at system generation for you installation. Other ddnames may have been specified at your installation; if they have, your JCL statements must use them in place of those given above. In a similar manner the IBM-supplied cataloged procedures PLIXCG and PLIXG use names as shown above; your systems programmer will have to modify these procedures if the names at your installation are different.

PRIMARY INPUT (SYSLIN)

Primary input to the loader must be a standard data set defined by a DD statement with the name SYSLIN; this data set must have consecutive organization. The input can comprise one or more object modules, one or more load modules, or a mixture of object modules and load modules.

For a PL/I program the primary input is usually a data set containing an object module produced by the compiler. This data set may be on magnetic tape or on a direct-access device, or you can include it in the input job stream. In all cases the input must be in the form of 80-byte F-format records.

The IBM-supplied cataloged procedure PLIXCG includes the DD statement:

```
//SYSLIN DD DSN=&&LOADSET,DISP=(OLD,DELETE)
```

This statement specifies that the data set &&LOADSET is temporary. If you want to modify this statement you must refer to it by the qualified ddname GO.SYSLIN.

The IBM-supplied cataloged procedure PLIXG does not include a DD statement for the input data set; you must always supply one specifying the characteristics of your input data set using the qualified ddname GO.SYSLIN.

AUTOMATIC CALL LIBRARY (SYSLIB)

Unless you specify otherwise, the loader will normally attempt to resolve external references by automatic library call. The automatic call library (SYSLIB), and how to specify it, is described under "Linkage Editor" on page 66.

LISTING (SYSLOUT)

The loader generates a listing that includes a module map (if you have specified the MAP option) and, if errors have been detected during processing, messages referring to these. The information that can appear is described under "Listing Produced by the Loader" on page 98.

You must define the data set in which you want this listing to be stored by a DD statement with the name SYSLOUT and it must have consecutive organization. Although the listing is usually printed it can be stored on any magnetic-tape or direct-access device. For printed output the following DD statement will suffice:

```
//SYSLOUT DD SYSOUT=A
```

LISTING (SYSPRINT)

As well as the information listed in the data set defined by the DD statement with the name SYSLOUT certain information produced by the loader is always stored in the data set defined by the DD statement with the name SYSPRINT. This data set, which must have consecutive organization, holds messages that refer to errors that have occurred during execution of your program, as well as the results produced by your program. The information that may appear is described under "Listing Produced by the Loader" on page 98. For printed output the following DD statement will suffice:

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

EXAMPLES OF LOADER JCL

A sequence of job control language for the loader is shown in Figure 36 on page 95. A PL/I program has been compiled in a job step with the step name PLI; the resultant object module has been placed in the data set defined by the DD statement with the name SYSLIN. Because this module is to be loaded and executed in the same job as the compile step, this DD statement can use the backward reference, indicated by the asterisk, as shown. If the compile and load-and-go steps were in different jobs, the DD statement would have to specify a permanent data set, cataloged or uncataloged.

The IBM-supplied cataloged procedure PLIXCG includes a DD statement with the name SYSLIN in both the compile and load-and-go procedure steps; you do not need to specify this statement unless you want to modify it. The IBM-supplied

cataloged procedure PLIXG does not include a DD statement with the name SYSLIN; you must supply one, using the qualified name GO.SYSLIN.

Typical job control language statements for the loader are shown in Figure 37. The example illustrates how to include, in the input stream, both an object module for input to the loader, and data to be used by your program during execution.

```
//LOAD       JOB
             .
             .
             .
//STEP1      EXEC PGM=LOADER
//SYSLIN     DD DSN=*.PLI.SYSLIN,DISP=(OLD,DELETE)
//SYSLIB     DD DSN=SYS1.PLIBASE,DISP=SHR
//SYSLOUT    DD SYSOUT=A
//SYSPRINT   DD SYSOUT=A
```

Figure 36. Job Control Language for Load-and-Go

```
//LOAD       JOB
             .
             .
             .
//STEP1      EXEC PGM=LOADER
//SYSLIN     DD DSN=OBJECT,UNIT=SYSSQ,VOL=SER=nnnnnn,DISP=(OLD,KEEP)
//           DD DSN=MODLIB(MOD55),DISP=SHR
//           DD DDNAME=IN
//SYSLIB     DD DSN=SYS1.PLIBASE,DISP=SHR
//           DD DSN=PRIVLIB,DISP=SHR
//SYSLOUT    DD SYSOUT=A
//SYSPRINT   DD SYSOUT=A
//SYSIN        DD *
             .
             .
             .
     (insert here the object module to be included)
             .
             .
             .
/*
//GO.SYSIN     DD *
             .
             .
             .
     (insert here the execution data, if any)
/*
```

Figure 37. Object and Load Modules in Load-and-Go

The DD statement with the name SYSLIN and the two following unnamed DD statements define three data sets that are to be concatenated into one data set to be used as input to the loader. The first data set is named OBJMOD and contains an object module. This data set could be the output of the optimizing compiler that has just processed your PL/I program. The second data set is named MODLIB(MOD55) containing a load module that has been given the name MOD55 and stored in the library called MODLIB. The third data set is an object module defined by the DD statement with the name IN. This DD statement appears further on and has the asterisk notation that indicates that the data set defined by this statement follows in the input stream.

The DD statement with the name SYSLIB and the unnamed DD statement immediately following it define two data sets that are to be concatenated so that they can be searched for unresolved external references by automatic library call. The first data set is the PL/I resident library (SYS1.PLILIB) and the second is a private library called PRIVLIB.

OPTIONAL FACILITIES OF THE LOADER

The loader provides a number of optional facilities that are selected by including the appropriate keywords from the following list in the PARM parameter of the EXEC statement that invokes it:

```
CALL
EP
LET
MAP
PRINT
RES
SIZE
```

Code the PARM parameter as follows:

```
PARM='[loader-options] [/execution-options]
          [/pgmparm]'
```

where "loader-options" is a list of loader options, "execution-options" is a list of execution-time options (as described in "Execution-Time Options" on page 31), and "pgmparm" is a parameter to be passed to the main procedure of the PL/I program to be executed. In the examples given below, the program parameter is referred to as PP.

If loader-options and either execution-options or a program parameter (or both) occur in the PARM parameter, the loader-options are given first and are separated from the execution-options for program parameter by a slash. If there are loader-options but no execution-options or program parameter, the slash is omitted, but if there are only execution-options or program parameters, the slash must be coded. If there is more than one option, the option keywords are separated by commas.

The PARM field can have one of three formats:

1. If the special characters / or = are used, the field must be enclosed in single quotes. For example:

   ```
   PARM='MAP,EP=FIRST/PP'
   PARM='MAP,EP=FIRST'
   PARM='/PP'
   ```

2. If these characters are not included, and there is more than one loader option, the options must be enclosed in parentheses. For example:

   ```
   PARM=(MAP,LET)
   ```

3. If these characters are not included, and there is only one loader option, neither quotes nor parentheses are required. For example:

   PARM=MAP

To override the PARM parameter options specified in a cataloged procedure, you must refer to the PARM parameter by the qualified name PARM.procstepname. For example: PARM.GO

The loader options are of two types:

1. Simple pairs of keywords: a positive form (for example, CALL) that requests a facility, and an alternative negative form (for example, NOCALL) that rejects that facility. CALL, LET, MAP, PRINT, and RES are of this type.

2. Keywords that permit you to assign a value to a function (for example, SIZE=10K). EP and SIZE are of this type.

The loader options are described in the following sections, in alphabetic order.

CALL OPTION

The CALL option specifies that the loader will attempt to resolve external references by automatic library call. To preserve compatibility with the linkage editor, the negative form of this option can be specified as NCAL as well as by NOCALL.

EP OPTION

The EP option specifies the entry point name of the program that is to be executed. The syntax of the EP option is:

  EP=name

where "name" is an external name. If all input modules are load modules you must specify EP=PLISTART.

LET OPTION

The LET option specifies that the loader will try to execute the problem program even if a severity 2 error has been found.

MAP OPTION

The MAP option specifies that the loader is to print a map of the load module giving the relative locations and lengths of control sections in the module. You must specify the data set defined by the DD statement with the name SYSLOUT to have this map printed. The module map is described in "Listing Produced by the Loader" on page 98.

PRINT OPTION

The PRINT option specifies that the data set defined by the DD statement with the name SYSLOUT is to be used for messages, the module map, and other loader information.

RES OPTION

The RES option specifies that the loader will attempt to resolve external references by a search of the link-pack area of main storage. This search will be made after the primary input to the loader has been processed but before the data set defined by the DD statement with the name SYSLIB is opened.

SIZE OPTION

The SIZE option specifies the amount of main storage, in bytes, to be allocated to the loader. The syntax of the SIZE option is:

SIZE=yyyyyy
: specifies that yyyyyy bytes of main storage are to be allocated to the loader.

SIZE=yyyK
: specifies that yyyK bytes of main storage are to be allocated to the loader (1K=1024).

The values can be enclosed, optionally, in parentheses.

LISTING PRODUCED BY THE LOADER

The loader can provide a listing on the SYSLOUT data set; the SYSPRINT data set is used by the problem program. The contents of each is given in Figure 38.

Data Set	Contents
SYSLOUT	Loader explanatory messages and diagnostic messages, and optionally, a module map.
SYSPRINT	PL/I execution-time messages, and problem program output.

Figure 38. Contents of SYSLOUT and SYSPRINT Data Sets

The SYSLOUT listing is described here; the SYSPRINT listing is described under "Listing (SYSPRINT)" on page 10.

The items in the SYSLOUT listing appear in the following sequence:

1. Statement identifying the loader.
2. Module map (if specified).
3. Explanatory, error, or warning messages.
4. Diagnostic messages.

MODULE MAP

If the MAP option is specified, a module map is printed in the SYSLOUT listing. The map lists all the control sections in the load module and all the entry point names (other than the first) in each control section. The information for each reference is:

- The control section or entry point name.
- An asterisk, if the control section is included by library call.

- An identifier, as follows:

  SD Section definition: the name of the control section.

  LR Label reference: identifying an entry point in the control section other than the primary entry point.

  CM Common area: an external file, or a non-string element variable declared STATIC EXTERNAL.

- Absolute address of the control section or entry point.

Each reference is printed left to right across the page and starts at a preset position. This gives the impression that the references are arranged in columns, but the correct way to read the map is line by line, across the page, not down each column.

The module map is followed by a similar listing of the pseudoregisters. The identifier used here is PR, and the address is the offset from the beginning of the pseudoregister vector (PRV). The total length of the PRV is given at the end.

The total length of the module to be executed, and the absolute address of its primary entry point, are given after the explanatory messages and before the diagnostic messages.

EXPLANATORY AND DIAGNOSTIC MESSAGES

The loader generates messages describing errors or conditions, detected during processing by the loader, that may lead to error. The format of these messages is given under "Diagnostic Message Directory" on page 77.

When the module to be executed has been processed, the loader prints out in full all the messages referred to above. The text of each message, an explanation, and any recommended programmer response, are given in OS/VS Message Library: Linkage Editor and Loader Messages.

CHAPTER 4. DATA SETS AND FILES

This chapter describes briefly the nature and organization of data sets, the data management services provided by the operating system, the record formats acceptable for auxiliary storage devices, and the way in which data sets are associated with PL/I files. It also describes some ENVIRONMENT options used in file declarations to describe the data set to PL/I. Methods of creating and accessing data sets are given in Chapter 5, "Defining Data Sets for Stream Files" on page 134, Chapter 6, "Using Consecutive, Indexed, Regional, and Teleprocessing Data Sets" on page 149, and Chapter 7, "Using VSAM Data Sets from PL/I" on page 222.

Chapter 7, "Using VSAM Data Sets from PL/I" on page 222 describes VSAM data sets. These differ significantly from other data set types; VSAM users will find that much of the information in this chapter is irrelevant.

DATA SETS

A data set is any collection of data that can be created by a program and accessed by the same or another program. A data set may be a deck of punched cards, it may be a series of items recorded on magnetic tape, or it may be recorded on a direct-access device (as well as being input from, or output to, your terminal). A printed listing produced by a program is a data set, but it cannot be accessed by a program.

A volume is a physical unit of auxiliary storage (for example, a reel of magnetic tape or a disk pack) that can be written on or read by an input/output device; a serial number identifies each volume (other than a magnetic-tape volume either without labels or with nonstandard labels).

A magnetic-tape or direct-access volume can contain more than one data set; conversely, a single data set can span two or more magnetic-tape or direct-access volumes.

DATA SET NAMES

A data set on a direct-access device must have a name so that the operating system can refer to it. If you do not supply a name, the operating system will supply a temporary one. A data set on a magnetic-tape device must have a name if the tape has IBM standard labels (see "Labels" on page 105). Names can be unqualified, qualified, temporary, or generation names, as described in your JCL manual. Data sets on punched cards, paper tape, unlabeled magnetic tape, or nonstandard labeled magnetic tape do not have names.

You can place the name of a data set, with information identifying the volume on which it resides, in a catalog. Such a data set is termed a cataloged data set. To catalog a data set, use the CATLG subparameter of the DISP parameter of the DD statement. To retrieve a cataloged data set, you need only specify the name of the data set and its disposition. The operating system searches the catalog for information associated with the name and uses this information to request the operator to mount the volume containing your data set.

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. A block could also contain a prefix field of up to 99 bytes in length depending on the information interchange code (ASCII or EBCDIC) in which the data is recorded (see "Information Interchange Codes"). Specify the block size in the BLKSIZE parameter of the DD statement or in the BLKSIZE option of the ENVIRONMENT attribute.

A record is the unit of data transmitted to and from a program. 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.

If a block contains two or more records, the records are said to be blocked. Blocking conserves storage space in a volume because it reduces the number of interblock gaps, and it may increase efficiency by reducing the number of input/output operations required to process a data set. Records are blocked and deblocked by the data management routines.

Specify the record length in the LRECL parameter of the DD statement or in the RECSIZE option of the ENVIRONMENT attribute.

INFORMATION INTERCHANGE CODES: The normal code in which data is recorded is the Extended Binary Coded Decimal Interchange Code (EBCDIC), although source input can optionally be coded in Binary Coded Decimal (BCD). However, for magnetic tape only, the system accepts data recorded in the American Standard Code for Information Interchange (ASCII). Use the ASCII and BUFOFF options of the ENVIRONMENT attribute if you are reading or writing data sets recorded in ASCII.

A prefix field up to 99 bytes in length may 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 options used for ASCII data sets, see "Consecutive Data Sets" on page 149.

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 must be one of the following:

- Fixed-length
- Variable-length
- Undefined-length

Records can be blocked if required, but only fixed-length and variable-length records are deblocked by the system; undefined-length records must be deblocked by your program.

Fixed-Length Records

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

F	Fixed-length, unblocked
FB	Fixed-length, blocked
FS	Fixed-length, unblocked, standard
FBS	Fixed-length, blocked, standard

In a data set with fixed-length records, as shown in Figure 39, 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 may be truncated). If the records are unblocked, each record constitutes a block.

Unblocked Records (F-format):

Blocked Records (FB-format):

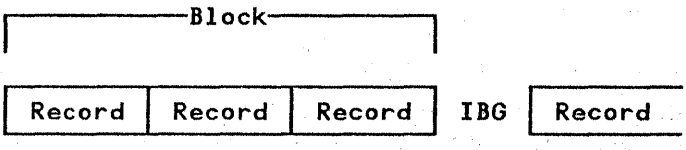


Figure 39. Fixed-length Records

Because it can base blocking and deblocking on a constant record length, the operating system can process fixed-length records faster than it can variable-length records.

The use of "standard" (FS-format and FBS-format) records further optimizes the sequential processing of a data set on a direct-access device. A standard format data set must contain fixed-length records and must have no embedded empty tracks or short blocks (apart from the last block). With a standard format data set, the operating system can predict whether the next block of data will be on a new track and, if necessary, can select a new read/write head in anticipation of the transmission of that block. A PL/I program never places embedded short blocks in a data set with fixed-length records. A data set containing fixed-length records can be processed as a standard data set even if it is not created as such, providing it contains no embedded short blocks or empty tracks.

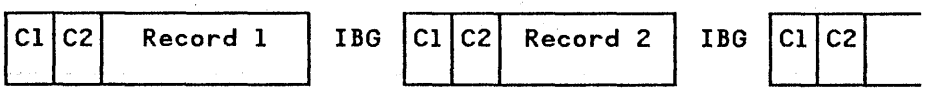
Variable-Length Records

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

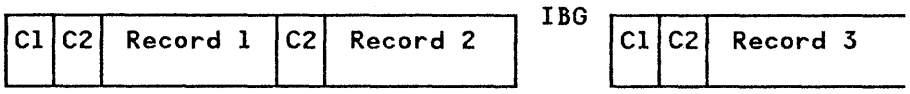
V	Variable-length, unblocked
VB	Variable-length, blocked
VS	Variable-length, unblocked, spanned
VBS	Variable-length, blocked, spanned
D	Variable-length, unblocked, ASCII
DB	Variable-length, blocked, ASCII

V-format permits both variable-length records and variable-length blocks. The first 4 bytes of each record and of each block contain control information for use by the operating system (including the length in bytes of the record or block). Because of these control fields, variable-length records cannot be read backward. Illustrations of variable-length records are shown in Figure 40.

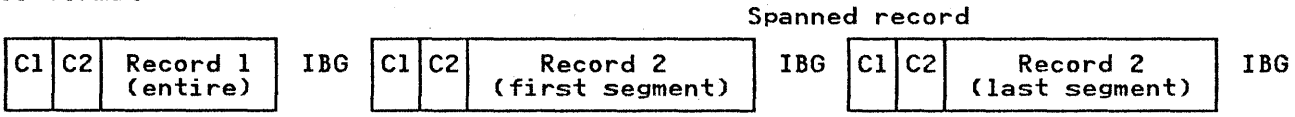
V-format:



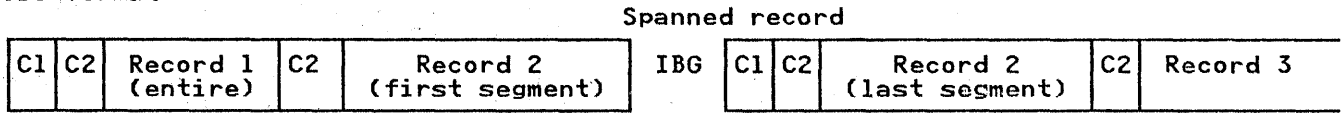
VB-format:



VS-format:



VBS-format:



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

Figure 40. Variable-Length Records

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 the first 4 bytes of each record contain record control information.

SPANNED RECORDS: A spanned record is a variable-length record in which the length of the record can exceed the size of a block. If this occurs, the record is divided into segments and accommodated in two or more consecutive blocks by specifying the record format as either VS or VBS. Segmentation and 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 combine optimum use of auxiliary storage with maximum efficiency of transmission.

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

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).

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 permits 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 permitted means of access to the data. The three 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
Indexed sequential	INDEXED
Direct	REGIONAL

The compiler recognizes a fourth type, teleprocessing, by the file attribute TRANSIENT.

A fifth type, partitioned, has no corresponding PL/I organization. VSAM also provides a number of alternatives.

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 may be selected for direct-access devices. Paper tape, punched cards, terminal, and printed output are sequentially organized.

---

[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.

An indexed sequential (or INDEXED) data set must reside on a direct-access volume. An index or set of indexes maintained by the operating system gives the location of certain principal records. This permits direct retrieval, replacement, addition, and deletion of records, as well as sequential processing.

A direct (or REGIONAL) data set must reside on a direct-access volume. The records within the data set can be organized in three ways: REGIONAL(1), REGIONAL(2), and REGIONAL(3); in each case, the data set is divided into regions, each of which contains one or more records. A key that specifies the region number and, for REGIONAL(2) and REGIONAL(3), identifies the record, permits direct-access to any record; sequential processing is also possible.

A teleprocessing data set (associated with a TRANSIENT file in a PL/I program) must reside in storage. Records are placed in physical sequence.

In a partitioned data set, independent groups of sequentially organized data, each called a member, reside in a direct-access data set. The data set includes a directory that lists the location of each member. Partitioned data sets are often called libraries. The compiler includes no special facilities for creating and accessing partitioned data sets. Each member can be processed as a CONSECUTIVE data set by a PL/I program. The use of partitioned data sets as libraries is described under Chapter 8, "Libraries of Data Sets" on page 264.

LABELS

The operating system uses labels to identify magnetic-tape and direct-access volumes, and to store data set attributes (for example, record length and block size). The attribute information must originally come from a DD statement or from your program. Once the label is written you need not specify the information again.

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, termed a data set control block (DSCB), for each data set stored on the volume.

DATA DEFINITION (DD) STATEMENT

A data definition (DD) statement is a job control statement that defines a data set to the operating system, and is a request to the operating system for the allocation of input/output resources. Each job step must include a DD statement for each data set that is processed by the step.

Your JCL manual describes the syntax of job control statements. The operand field of the DD statement can contain keyword parameters that describe the location of the data set (for example, volume serial number and identification of the unit on which the volume will be mounted) and the attributes of the data itself (for example, record format).

The DD statement enables you to write PL/I source programs that are independent of the data sets and input/output devices they will use. You can modify the parameters of a data set or process different data sets without recompiling your program; for example, you can cause a program that originally read punched cards to accept input from magnetic tape by changing the DD statement.

The following paragraphs describe the relationship of some operands of the DD statement to your PL/I program.

The LEAVE and REREAD options of the ENVIRONMENT attribute allow you to use the DISP parameter to control the action taken when the end of a magnetic-tape volume is reached or when a magnetic-tape data set is closed. The LEAVE and REREAD options are described under "Consecutive Data Sets" on page 149, and are also described under "CLOSE Statement" in the OS and DOS PL/I Language Reference Manual.

Use of the Conditional Subparameters

If you use the conditional subparameters of the DISP parameter for data sets processed by PL/I programs, the step abend facility must be used. The step abend facility is obtained as follows:

1. The ERROR condition should be raised or signaled whenever the program is to terminate execution after a failure that requires the application of the conditional subparameters.
2. The resident library subroutine IBMBEER must be changed to return a nonzero return code. The method of doing this is described in OS PL/I Optimizing Compiler Installation Guide for MVS.

DATA SET CHARACTERISTICS: The DCB (data control block) parameter of the DD statement allows you to describe the characteristics of the data in a data set, and the way it will be processed, at execution time. Whereas the other parameters of the DD statement deal chiefly with the identity, location, and disposal of the data set, the DCB parameter specifies information required for the processing of the records themselves. The subparameters of the DCB parameter are described in your JCL manual. For DCB use, see "Data Control Block" on page 117.

The DCB parameter contains subparameters that describe:

- The organization of the data set and how it will be accessed (CYLOFL, DSORG, LIMCT, NCP, NTM, and OPTCD subparameters)
- Device-dependent information such as the recording technique for magnetic tape or the line spacing for a printer (CODE, DEN, FUNC, MODE, OPTCD=J, PRTSP, STACK, and TRTCH subparameters)
- The record format (BLKSIZE, KEYLEN, LRECL, RECFM, and RKP subparameters)
- The number of buffers that are to be used (BUFNO subparameter)
- The printer or card punch control characters (if any) that will be inserted in the first byte of each record (RECFM subparameter)

You can specify BLKSIZE, BUFNO, LRECL, KEYLEN, NCP, RECFM, RKP, and TRKOFL (or their equivalents) in the ENVIRONMENT attribute of a file declaration in your PL/I program instead of in the DCB parameter.

You cannot use the DCB parameter to override information already established for the data set in your PL/I program (by the file

attributes declared and the other attributes that are implied by them). DCB subparameters that attempt to change information already supplied are ignored.

An example of the DCB parameter is:

```
DCB=(RECFM=FB,BLKSIZE=400,LRECL=40)
```

which specifies that fixed-length records, 40 bytes in length, are to be grouped together in a block 400 bytes long.

AUXILIARY STORAGE DEVICES

The following paragraphs summarize the salient operational features of various types of auxiliary storage devices.

IBM 2520 AND 2540 CARD READER AND PUNCH

Both the card reader and card punch accept F-format, V-format, and U-format records; the control bytes of V-format records are not punched. Any attempt to block records is ignored.

Each punched card corresponds to one record; you should therefore restrict the maximum record length to 80 bytes (EBCDIC mode) or 160 bytes (column-binary mode). To select the mode, use the MODE subparameter of the DCB parameter of the DD statement; if you omit this subparameter, EBCDIC is the default. (The column-binary mode increases the packing density of information on a card, punching 2 bytes in each column. Only 6 bits of each byte are punched; on input, the 2 high-order bits of each byte are set to zero; on output, the 2 high-order bits are lost.) The IBM 2540 Card Read Punch has five stackers into which cards are fed after reading or punching. Two stackers accept only cards that have been read, and two others accept only those that have been punched; the fifth (center) stacker can accept either cards that have been read or those that have been punched. The two stackers in each pair are numbered 1 and 2 and the center stacker is numbered 3, as shown in Figure 41.

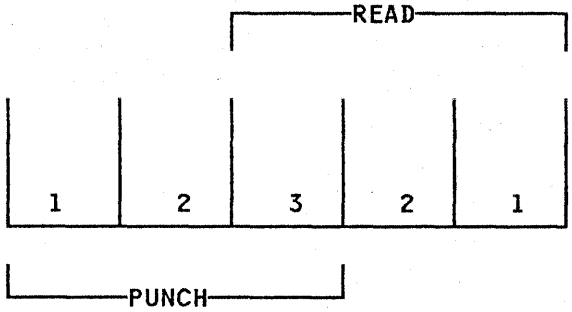


Figure 41. IBM 2540 Card Read Punch: Stacker Numbers

The IBM 2520 Card Read Punch has two stackers, into which cards can be read or punched. The IBM 2501 Card Reader has only one stacker.

Cards are normally fed into the appropriate stacker 1 after reading or punching. You can use the STACK subparameter of the DCB parameter of the DD statement to select an alternative stacker for reading or punching. For punching only, you can select the stacker dynamically by inserting an American National Standard or machine code control character in the first byte of

each record; you must indicate which you are using in the RECFM subparameter of the DD statement or in the ENVIRONMENT option. The control character is not punched.

IBM 3505 AND 3525 CARD READER AND PUNCH

The IBM 3505 Card Reader and the IBM 3525 Card Punch are functionally separate and operate independently of each other.

The 3505 reads 80-column cards, and provides, in addition to normal card reading, the following facilities:

- Optical Mark Read (in EBCDIC or column binary mode)
- Read Column Eliminate (in EBCDIC or column binary mode)
- Stacker selection

The 3525 is basically an 80-column card punch, and can have the following additional facilities:

- Card reading facilities that optionally include:
  - Reading in EBCDIC or column binary mode
  - Read Column Eliminate
- Card punching in EBCDIC or column binary mode
- Card printing facilities that include either:
  - Two-line printing, or
  - Multiline printing (up to 25 lines)
- Punch Interpretation
- Stacker selection

The various operations of the 3505 and the 3525 are described in the following sections. In general, the operations to be performed are selected by the FUNC, MODE, and STACK subparameters of the DCB parameter. The formats of these subparameters are described in your JCL manual.

Basic Card Reading and Punching

Card reading or punching on a 3525 is selected by specifying DCB=(FUNC=R) for reading or DCB=(FUNC=P) for punching. If the FUNC subparameter is not specified, the default is FUNC=R for input files and FUNC=P for output files that do not have the PRINT attribute.

Apart from this function selection for the 3525, support for the 3505 as a simple card reader and the 3525 as a card reader or punch is identical to that for the 2540 described under "IBM 2520 and 2540 Card Reader and Punch" on page 107.

EBCDIC or Column Binary Modes

Cards processed by a 3505 or a 3525 can hold data coded in either EBCDIC or column binary mode. If EBCDIC is used, each card can contain up to 80 characters. If column binary mode is used, each card can contain up to 160 binary characters, two per card column. EBCDIC and column binary data cannot be intermixed.

In column binary mode, each card column holds two 6-bit characters. The first character appears in rows 12 through 3 on the card, and the second in rows 4 through 9. The binary values of characters are transmitted to successive bytes in main

storage. The 2 high-order bits of each byte are set to zero (these bits are not represented in the 6-bit code). The characters are transmitted in the order: first (top) character, second (bottom) character, and so on for each column in the card, from column 1 to column 80.

The details of the coding and conversion technique used for column binary data are left to the program designer. The TRANSLATE built-in function may provide a convenient method of converting data to or from column binary form.

Rules for using column binary mode are:

- The MODE subparameter of the DCB parameter must specify column binary (MODE=C).
- The PL/I file must have the RECORD attribute.
- The punch-interpret feature must not be used.
- The file must be either an input file or an output punch file. It cannot be a print file.
- A column binary output file must have a record size of 160 bytes.

Stacker Selection

The stacker selection feature is optionally available on the 3505 and is a standard feature on a 3525. There are two methods of selecting a stacker:

- The stacker can be selected permanently for all cards in the file. This method involves the STACK subparameter of the DCB parameter.
- For record-oriented data transmission to a 3525, the first byte of the record can contain a stacker control character to select the required stacker dynamically. The use of such codes is specified by the CTLASA or CTL360 ENVIRONMENT options.

Optical Mark Read

The optical mark read (OMR) feature is available only on the 3505 card reader. This feature enables preprinted or pencil-written marks on a punched card to be read as data. The following rules apply:

- Optical Mark Read is specified by MODE=EO (EBCDIC mode) or MODE=CO (column binary mode) in the DCB parameter.
- The associated PL/I file must have the RECORD and INPUT attributes, and must not have the TOTAL attribute.
- Records must be F-format with a RECSIZE of 80 (EBCDIC mode) or 160 (column binary mode).
- Up to 40 columns of EBCDIC data or 80 characters of column binary data can be read optically from a single card. Optical and punched data can be read from the same card although there are some restrictions, given below, on how the data is recorded on the card.
- Optical mark data can appear only in alternate card columns and must be separated by blank columns. Optical mark and punched hole columns must also be separated by at least one blank column. When the record is read in, the data is compressed by removing the blank column following each optical mark column, and the record is padded with blanks.

- The columns containing optically-readable marks must be specified to the program at execution time by a format descriptor card. This card must be the first card in the deck of cards to be read by the file each time the program is run. Operating procedures for running jobs that use OMR should ensure that this point is not overlooked.

- The OMR descriptor card has the following format:

  ```
  FORMAT (n1,n2),(n3,n4)...
  ```

  where n1 is the first column in a group to be read in OMR mode, n2 is the last column in the group, n3 is the first column in the next group, n4 is the last column in this group, and so on. Remember that only every other column between n1 and n2 or n3 and n4 can be read in OMR mode. A maximum of 40 columns of OMR data can be accommodated on an 80-column card. n1 and n2 (and similarly n3 and n4) must be either both even or both odd, and n3 must be at least 2 greater than n2.

  The format descriptor record must begin in column 2 and can continue through column 71. If a continuation is required, punch any character in column 72 and start the continuation in column 16 of the following card.

  A blank must follow the keyword FORMAT. Operands must be separated by commas. For example:

  ```
  FORMAT (1,9),(70,80)
  ```

  This specifies that columns 1 to 10 and 70 to 80 are reserved for OMR use and, of these, columns 1, 3, 5, 7, 9, 70, 72, 74, 76, 78, and 80 will be scanned for optical mark data.

- Column 1 of the card always corresponds with the first byte of the data in main storage. Consequently, if an optical mark appears in column 2, column 1 must be blank and the first byte of storage will also be blank.

- If a marginal mark, weak mark, or poor erasure is detected on a column, the corresponding byte and the last byte of the record are set to X'3F'. The TRANSMIT condition is raised once only for all errors found in a card. The card itself is stacked in the alternative stacker to that normally used by the file.

- When an optical mark read file is closed, the last card is fed and stacked in the same stacker as the previous card. This feed operation resets the device into unformatted mode, ready for the next file opening.

- Optical Mark Read is not supported on SYSIN. The 3505 must be allocated exclusively to the user's job by specifying the device type of the unit address in the UNIT parameter of the DD statement.

- When a file is opened for optical mark reading, the value of the BUFFERS option (for BUFFERED files) or the NCP option (for UNBUFFERED files) is set to 1.

Read Column Eliminate

The Read Column Eliminate (RCE) feature is optionally available on the 3505 and on a 3525 with card reading facilities. This feature permits the selective reading of card columns. The columns to be ignored when the card is read are specified in a format descriptor card. The ignored columns are replaced by blanks in EBCDIC mode or zeros in column binary mode before the record is transmitted.

The following rules apply:

- Read Column Eliminate is specified by MODE=ER (EBCDIC mode) or MODE=CR (column binary mode) in the DCB parameter.
- An RCE format descriptor card must be supplied. This card must be the first card in the deck of cards to be read by the program each time it is executed. Operating procedures for running jobs that use RCE should ensure that this point is not overlooked.
- The RCE descriptor card has the following format:

  ```
  FORMAT (n1,n2),(n3,n4)...
  ```

  where n1 is the first column in a group of columns to be ignored and n2 is the last column in the group, n3 is the first column in the next group to be ignored, n4 is the last column in this group, and so on.

  The format descriptor card must begin in column 2 and continue through to column 71. If a continuation is required, punch any character in column 72 and start the continuation in column 16 of the following card.

  A blank must follow the keyword FORMAT. Operands must be separated by commas. For example:

  ```
  FORMAT (20,30),(52,76)
  ```

  This specifies that columns 20 through 30 and columns 52 through 76 are to be ignored when the card is read.
- The PL/I file can have either the STREAM or the RECORD attribute. Records must be F-format.
- When an RCE file is closed, a card feed operation is executed by the reader. If several files are to be read consecutively — either for successive programs in a single batch, or for several files in a single program — a nondata card must separate the files.
- Read Column Eliminate is not supported on SYSIN. The 3505 or 3525 must be allocated exclusively to the user's job by specifying the device type of the unit address in the UNIT parameter of the DD statement.

Punch Interpret

A single file can be used to punch and interpret cards by specifying DCB=(FUNC=I). Cards are punched normally, and the same data is printed on lines 1 and 3 of the card. The first 64 characters are printed on line 1; the remaining 16 characters are right-justified on line 3.

A punch interpret file may have the STREAM or RECORD and the BUFFERED or UNBUFFERED attributes. Records must be F-format, with a record size of 80, or 81 if control characters are being used for stacker selection.

Printing on Cards

The card printing feature of the 3525 is available in two forms:

- Two-line printing
- Multiline printing (up to 25 lines)

Printing can be performed either as the only operation on the card, or as one of a number of operations on the same card. The following rules apply to print-only files. The additional requirements for printing after reading or punching a card are described under "Multiple Operations" on page 112.

- The FUNC subparameter of the DCB parameter must specify "W" if the 3525 has the multiline print feature, or "WT" if it has the two-line print feature. If FUNC is omitted, FUNC=W is defaulted for PL/I PRINT files.

- The PL/I file may have either the RECORD or the STREAM attribute.

- The maximum number of characters that can be printed on each line is 64. You must ensure that this limit is not exceeded; in particular, on PRINT files, LINESIZE should not exceed 64 or data will be lost.

- If the 3525 has the two-line print feature, and is used by a file with the PRINT attribute or by a file using CTLASA or CTL360 control characters, you must ensure that no attempt is made to print on any line other than lines 1 and 3. Such an attempt will terminate the program without raising the PL/I ERROR condition. If a PRINT file is used, and a PAGESIZE of more than 3 is specified, the page size is set to 3 when the file is opened.

  If the file is a non-PRINT file, and control characters are not used, records are printed on lines 1 and 3.

- If a 3525 with the multiple print feature is used, the file should have a maximum page size of 25. If a PAGESIZE of greater than 25 is specified on a PRINT file, the page size is set to 25 when the file is opened. Whatever the page size, a PUT PAGE statement for a PRINT file will always cause the file to be positioned at line 1 of the next card. Any attempt to print beyond line 25 will terminate the program without raising the PL/I ERROR condition.

- All the American National Standard control characters can be used, with the exception of "+" (suppress space before printing). The use of the "+" control character, or of SKIP(0) on a PRINT file, will terminate the program without raising the PL/I ERROR condition.

  Odd-numbered lines on a card can be reached using "skip to channel" control characters, with channel numbers being defined as:

  ```
  channel number = (line number + 1)/2
  ```

  Only channels 1 through 12 are valid. Other lines can be reached by using "space and print" control characters. All lines can be reached by executing sufficient WRITE or PUT operations.

Multiple Operations

Two or three files may be used in association with each other to enable more than one of the operations "read," "punch," and "print" to be performed on a single card during one pass through a 3525. A DD statement is required for each operation that the device is to perform, and the association of these data sets is specified by means of the unit affinity (AFF) parameter, together with the FUNC subparameter of the DCB parameter.

For example, for a set of files that are to perform the operations read-punch-print, the association of the data sets and the set of operations is specified as follows:

```
//CARDIN DD UNIT=3525,DCB=(FUNC=RPW)
//PUNCH  DD UNIT=AFF=CARDIN,
//          DCB=(FUNC=RPW)
//PRINT  DD UNIT=AFF=PUNCH,
//          DCB=(FUNC=RPWX)
```

Valid FUNC options are listed in your JCL manual. Note that the FUNC option must specify the complete set of associated

operations. "X" must be added to the FUNC option of the print data set. If the 3525 has the two-line print feature, "T" must also be coded on the FUNC option of the print data set.

The following rules apply to multiple operations:

- All the device-associated files must have the RECORD attribute, and must be all BUFFERED or all UNBUFFERED. None of the files can have the TOTAL option. Records must be F-format.
- If stacker selection is required, it can only be specified on the punch file, if there is one. Either stacker-select control characters or static stacker selection by means of the STACK subparameter can be used.
- An associated data set cannot be allocated to SYSIN or SYSPRINT. The 3525 must be allocated exclusively to your job by specifying the device type of the unit address in the UNIT parameter of the DD statement.
- Data delimiter cards should not be punched or printed on, or the first card of the following job will be lost.

Details of how to open and close associated files, and of the sequences of operations that can be performed, are given in the OS and DOS PL/I Language Reference Manual.

Data Protection

To avoid erroneous punching into card columns that already contain data, a "data protection" option can be used on a punch file which is in association with a read file. Data protection is specified by a "D" in the FUNC option of the DD statement for the punch data set. You must provide an 80-byte data protection image (DPI) and link-edit it into SYS1.IMAGELIB with a member name of the form FORMxxxx. The DPI contains blanks in columns that are to be protected, and any alphameric character in columns that can be punched. An assembler language program is used to link-edit the DPI. For example:

```
//UP EXEC ASMFCL
//ASM.SYSIN DD *
FORMDPI  CSECT
         DC  X'40'      (protected column)
         DC  X'40'      (protected column)
         DC  C'3456789A'  (punch columns)
         DC  70X'40'  (protected columns)
         END
/*
//LKED.SYSLMOD DD DISP=OLD,
//         DSNAME=SYS1.IMAGELIB(FORMxxxx)
```

A particular DPI is selected by means of the FCB parameter of the DD statement for the punch file. For example:

```
//PUNCH  DD  UNIT=AFF=CARDIN,
//           DCB=(FUNC=RPWD),
//           FCB=xxxx
```

Data protection cannot be specified for column binary cards.

PAPER TAPE READER

The paper tape reader accepts F-format and U-format records; each U-format record is followed by an end-of-record character. Use the CODE subparameter of the DCB parameter of the DD statement to request translation of data from one of the six standard paper-tape codes to EBCDIC. Any character found to have a parity error is not transmitted.

LINE PRINTERS

The printer accepts F-format, V-format, and U-format records; the control bytes of V-format records are not printed. Each line of print corresponds to one record; you should therefore restrict your record length to the length of one printed line. Any attempt to block records is ignored.

You can use the PRTSP subparameter of the DCB parameter of the DD statement to request the line spacing of your output, or you can control the spacing dynamically by inserting an American National Standard or a machine-code print control character in the first byte of each record; you must indicate which you are using in the RECFM subparameter of the DD statement or in the ENVIRONMENT option. The control character is not printed. If you do not specify the line spacing, single spacing (no blanks between lines) is the default.

3800 PRINTING SUBSYSTEM

The IBM 3800 Printing Subsystem can be used in a manner that is compatible with IBM line printers. However, it can do more than line printers. For information on using its added capabilities, see your IBM 3800 Printing Subsystem Programmer's Guide.

MAGNETIC TAPE

Magnetic-tape devices accept ASCII, fixed-length, variable-length, and undefined-length records for both 9-track and 7-track magnetic tape, with the one exception that 7-track magnetic tape will not accept variable-length records unless the data conversion feature is available. (The data in the control bytes of variable-length records is in binary form; in the absence of the data conversion feature, only 6 of the 8 bits in each byte are transmitted to 7-track tape.)

Nine-track magnetic tape is used in IBM operating systems, but some 2400 series magnetic-tape drives incorporate features that facilitate reading and writing 7-track tape. The translation feature changes character data from EBCDIC (8-bit code) to BCD (the 6-bit code used on 7-track tape) or vice versa. The data conversion feature treats all data as if it were in the form of a bit string, breaking the string into groups of 8 bits for reading into main storage, or into groups of 6 bits for writing on 7-track tape; the use of this feature precludes reading the tape backward. To use translation or data conversion, include the TRTCH (tape recording technique) subparameter in the DCB parameter of the DD statement.

The maximum recording density available depends on the model number of the tape drive. You can use the subparameter DEN (density) of the DD statement to specify the recording density.

When a data check occurs on a magnetic-tape device with short length records (12 bytes on a read and 18 bytes on a write), these records will be treated as noise.

DIRECT-ACCESS DEVICES

Direct-access devices accept fixed-, variable-, and undefined-length records.

The storage space on these devices is divided into conceptual cylinders and tracks. A cylinder is usually the amount of space that can be accessed without movement of the access mechanism, and a track is that part of a cylinder that is accessed by a single read/write head. For example, an IBM 3380 Direct Access Storage device has 15 recording surfaces, each of which has 885 concentric tracks; thus, it contains 885 cylinders, each of which includes 15 tracks.

When you create a data set on a direct-access device, you must always indicate to the operating system how much auxiliary storage the data set requires. Use the SPACE parameter of the DD statement to allocate space in terms of blocks, tracks, or cylinders. If you request space in terms of tracks or cylinders, bear in mind that space in a data set on a direct-access device is occupied not only by blocks of data, but by control information inserted by the operating system; if you use small blocks, the control information can result in a considerable space overhead.

OPERATING SYSTEM DATA MANAGEMENT

The compiler compiles each input or output statement in a PL/I program into machine instructions that request the operating system data management routines to perform the required input or output operation. (For more information on PL/I data management, see the OS PL/I Optimizing Compiler: Execution Logic manual.)

The data management routines create and maintain data set labels, indexes, and catalogs; they transmit data between main storage and auxiliary storage; they use the system catalog to locate data sets; and they request the operator to mount and demount volumes as required.

BUFFERS

The data management routines can provide areas of main storage, termed buffers, in which data can be collected before it is transmitted to auxiliary storage, or into which it can be read before it is made available to a program. The use of buffers permits the blocking and deblocking of records, and may allow the data management routines to increase the efficiency of transmission of data by anticipating the needs of a program. Anticipatory buffering requires at least two buffers; while the program is processing the data in one buffer, the next block of data can be read into another. Anticipatory buffering can only be used for data sets being accessed sequentially.

The operating system can further increase the efficiency of transmission in a program that involves many input/output operations by using chained scheduling. In chained scheduling, a series of read or write operations are chained together and treated as a single operation. For chained scheduling to be effective, at least three buffers are necessary. For more information on chained scheduling, see your Data Management Services Guide.

Chained scheduling should not be used for certain filetypes in multitasking programs. See OPTCD in your JCL manual.

Record-oriented data transmission has two modes of handling data:

- In move mode, you can process data by having the data moved into or out of the variable, either directly or via a buffer.
- In locate mode, you can process data while it remains in a buffer. The execution of a data transmission statement assigns to a pointer variable the location of the storage allocated to a record in the buffer. Locate mode is applicable only to BUFFERED files; the file must be either a SEQUENTIAL file or an INPUT or UPDATE file associated with a VSAM data set.

For more information, see "Processing Modes" in the OS and DOS PL/I Language Reference Manual.

ACCESS METHODS

The access methods used by the compiler are shown in Figure 42.

A queued access method deals with individual records, which it blocks and deblocks. The data management routines place a block of records in an input buffer and pass a single record to the program for each retrieval request from the program (that is, they deblock the records); each succeeding retrieval passes another record to the program. When the input buffer is empty, it is refilled with another block. Similarly, on output, the data management routines place records in an output buffer and, when the buffer is full, write out the records. Since the queued access technique brings records into main storage before they are requested, it can be used only for records that have been organized sequentially.

A basic access method moves blocks, not records. When a request is issued to retrieve a block, the data management routines pass a block of data to the program that issued the request; they do not deblock the records. Similarly, an output request transmits a block to auxiliary storage.

Access Method	Explanation
QSAM	Queued sequential access method. This combines the queued access technique with sequential organization.
QISAM	Queued indexed sequential access method. This combines the queued access technique with indexed sequential organization.
BSAM	Basic sequential access method. This combines the basic access technique with sequential organization.
BISAM	Basic indexed sequential access method. This combines the basic access technique with indexed sequential organization.
BDAM	Basic direct-access method. This combines the basic access technique with direct organization.
TCAM	Telecommunications access method. This combines the queued access technique with teleprocessing organization.
VSAM	Virtual Storage Access Method. This access method is described in Chapter 7, "Using VSAM Data Sets from PL/I" on page 222.

Figure 42. The Access Methods Used by the Compiler

The PL/I library subroutines use QSAM for stream-oriented data transmission; for record-oriented data transmission, they use the access methods shown in Figure 43 on page 117. They implement PL/I GET and PUT statements by transferring the appropriate number of characters from or to the buffers, and use GET and PUT macro instructions in the locate mode to fill or empty the buffers. (For paper tape, the library subroutines use move mode to permit translation of the transmitted characters before passing them to the PL/I program.)

Data Set Organization	File Attributes			Access Methods
CONSECUTIVE	SEQUENTIAL	INPUT OUTPUT UPDATE	BUFFERED	QSAM
			UNBUFFERED	BSAM
INDEXED	SEQUENTIAL	INPUT OUTPUT UPDATE	BUFFERED or UNBUFFERED	QISAM
	DIRECT	INPUT UPDATE	-	BISAM
REGIONAL	SEQUENTIAL	INPUT OUTPUT UPDATE	BUFFERED or UNBUFFERED	BSAM
	DIRECT	INPUT OUTPUT UPDATE	-	BDAM
TELEPROCESSING	TRANSIENT	INPUT OUTPUT	BUFFERED	TCAM
VSAM ESDS	SEQUENTIAL	INPUT OUTPUT UPDATE	BUFFERED or UNBUFFERED	VSAM
VSAM KSDS and RRDS	SEQUENTIAL	INPUT OUTPUT UPDATE	BUFFERED or UNBUFFERED	VSAM
	DIRECT	INPUT OUTPUT UPDATE	BUFFERED or UNBUFFERED	VSAM

Figure 43. Access Methods for Record-Oriented Data Transmission

DATA CONTROL BLOCK

A data control block (DCB), or an access method control block (ACB) for VSAM, is an area of storage that contains information about a data set and the volume that contains it. The data management routines refer to this information when they are processing a data set; no data set can be processed unless there exists a corresponding DCB. For a PL/I program, a PL/I library subroutine creates a DCB for the data set when a file is opened.

A data control block contains two types of information: data set characteristics and processing requirements. The characteristics include record format, record length, block size, and data set organization. The processing information may specify the number of buffers to be used, and it may include device-dependent information (for example, printer line spacing or magnetic tape recording density), and special processing options that are available for some data set organizations.

The information in the DCB comes from three sources:

- The file attributes declared implicitly or explicitly in the PL/I program
- The data definition (DD) statement for the data set
- If the data set exists, the data set labels

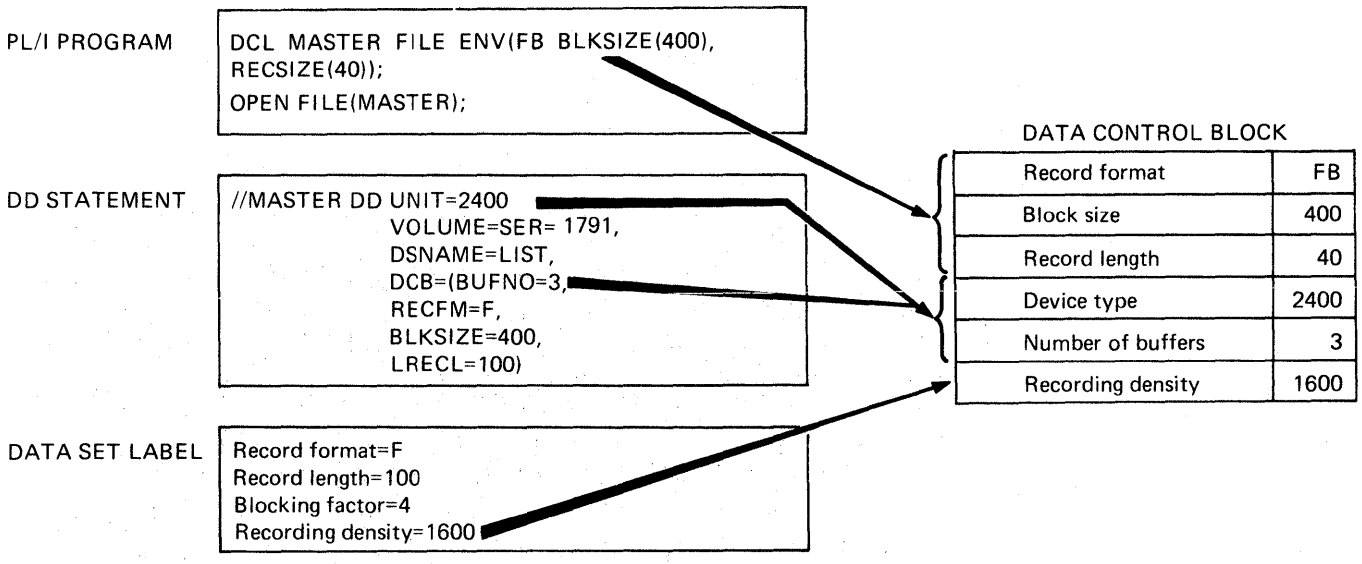
OPENING A FILE

The execution of a PL/I OPEN statement associates a file with a data set. This requires the merging of the information describing the file and the data set. If any conflict is detected between file attributes and data set characteristics, the UNDEFINEDFILE condition is raised.

Subroutines of the PL/I library create a skeleton data control block for the data set, and use the file attributes from the DECLARE and OPEN statements, and any attributes implied by the declared attributes, to complete the data control block as far as possible, as shown in Figure 44. They then issue an OPEN macro instruction, which calls the data management routines to check that the correct volume is mounted and to complete the data control block. The data management routines examine the data control block to see what information is still needed and then look for this information, first in the DD statement, and finally, if the data set exists and has standard labels, in the data set labels. For new data sets, the data management routines begin to create the labels (if they are required) and to fill them with information from the data control block.

Neither the DD statement nor the data set label can override information provided by the PL/I program; nor can the data set label override information provided by the DD statement.

When the DCB fields have been filled in from these sources, control returns to the PL/I library subroutines. If any fields have still not been filled in, the PL/I OPEN subroutine provides default information for some of them; for example, if LRECL has not been specified, it is now provided from the value given for BLKSIZE.



Note: Information from the PL/I program overrides that from the DD statement and the data set label.
Information from the DD statement overrides that from the data set label.

Figure 44. How the Operating System Completes the DCB

CLOSING A FILE

The execution of a PL/I CLOSE statement dissociates a file from the data set with which it was associated. The PL/I library subroutines first issue a CLOSE macro instruction and, when control returns from the data management routines, release the data control block that was created when the file was opened. The data management routines complete the writing of labels for new data sets and update the labels of existing data sets.

ASSOCIATING DATA SETS WITH FILES

With batch processing, the association of a file with a specific data set is accomplished using job control language, outside the PL/I program. At the time a file is opened, the PL/I file is associated with the name (ddname) of a data definition statement (DD statement), which defines a specific data set. The association is with the name of a DD statement, not with the name of the data set itself.

A ddname is associated with a PL/I file through the character value of the expression in the TITLE option of the OPEN statement.

If a file is opened implicitly, or if no TITLE option is included in the OPEN statement that explicitly opens the file, the ddname defaults to the file name. If the file name is longer than 8 characters, the default ddname is composed of the first 8 characters of the file name.

The character set of the job control language does not contain the break character (_). Consequently, this character cannot appear in ddnames. Do not use break characters among the first 8 characters of file names, unless the file is to be opened with a TITLE option with a valid ddname as its expression. The alphabetic extender characters $, @, and #, however, are valid for ddnames, but the first character must be one of the letters A through Z.

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

Consider the following statements:

```
1.  OPEN FILE(MASTER);

2.  OPEN FILE(OLDMASTER);

3.  READ FILE(DETAIL) ...;
```

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

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

1. //MASTER   DD ...

2. //OLDMASTE DD ...

3. //DETAIL   DD ...

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

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

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

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

Use of a file variable also allows a number of files to be manipulated 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 is used to read the three files A, B, and C, each of which may be associated with a different data set. The files A, B, and C remain open after the READ statement has been executed in each instance.

The following OPEN statement illustrates use of the TITLE option:

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

If this statement were executed, there must be a DD statement in the current job step with DETAIL1 as its ddname. It might appear, in part, as follows:

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

Thus, the data set DETAILA is associated with the file DETAIL through the ddname DETAIL1.

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

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

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

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

ASSOCIATING SEVERAL FILES WITH ONE DATA SET

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

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

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

CONCATENATING SEVERAL DATA SETS

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

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

When read from a PL/I program, the concatenated data sets need not be on the same volume, but the volumes must be on the same type of device, and the data sets must have similar characteristics (for example, block size and record format). You cannot process concatenated data sets backward.

THE ENVIRONMENT ATTRIBUTE

The ENVIRONMENT attribute of the PL/I file declaration specifies information about the physical organization of the data set associated with a file, and other related information. The information is contained in a parenthesized option list; the syntax is:

Syntax

ENVIRONMENT(option-list)

Abbreviation: ENV

The options may appear in any order, and are 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 7, "Using VSAM Data Sets from PL/I" on page 222).

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

Figure 45 on page 123 summarizes the ENVIRONMENT options and file attributes. Certain qualifications on their use are presented in the notes and comments for that figure. Those options that apply to more than one data set organization are described below. In addition, in the following chapters, each option is described with each data set organization to which it applies.

DATA SET ORGANIZATION OPTIONS

The options that specify data set organization are:

```
CONSECUTIVE
INDEXED
REGIONAL({1 | 2 | 3})
TP({M | R})
VSAM
```

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

If the data set organization option is not specified in the ENVIRONMENT attribute, a default is obtained when the file is opened:

- If the merged attributes from the DECLARE and OPEN statements do not include TRANSIENT, the default is CONSECUTIVE.
- If the attributes include TRANSIENT, the default is TP(M).

Types of File / File Attributes and ENVIRONMENT Options	Record										
	Sequential								Direct		
	Consecutive					Regional					
	Stream	Buffered	Unbuffered	Teleprocessing	Indexed	Buffered	Unbuffered	VSAM	VSAM	Indexed	Regional
FILE	I	I	I	I	I	I	I	I	I	I	I
INPUT[1]	D	D	D	D	D	D	D	D	D	D	D
OUTPUT	O	O	O	O	O	O	O	O	O	O	O
ENVIRONMENT	I	I	I	S	S	S	S	S	S	S	S
STREAM	D	—	—	—	—	—	—	—	—	—	—
PRINT[1]	O	—	—	—	—	—	—	—	—	—	—
RECORD	—	I	I	I	I	I	I	I	I	I	I
UPDATE[2]	—	O	O	—	O	O	O	O	O	O	O
SEQUENTIAL	—	D	D	—	D	D	D	D	D	—	—
BUFFERED	—	D	—	I	D	D	—	D	S	—	—
UNBUFFERED	—	—	S	—	—	—	S	S	D	D	D
BACKWARDS[3]	—	O	O	—	—	—	—	—	—	—	—
TRANSIENT	—	—	—	I	—	—	—	—	—	—	—
KEYED[4]	—	—	—	I	O	O	O	O	O	I	I
DIRECT	—	—	—	—	—	—	—	S	S	S	S
EXCLUSIVE	—	—	—	—	—	—	—	—	—	O	O

Key:

- I Must be specified or implied
- C Checked for VSAM
- D Default
- N Ignored for VSAM
- O Optional
- S Must be specified
- — Invalid

Comments	Attributes Implied
	FILE
	FILE
	FILE
	FILE
	FILE STREAM OUTPUT
	FILE
	FILE RECORD
	FILE RECORD
	FILE RECORD
	FILE RECORD
	FILE RECORD SEQUENTIAL INPUT
	FILE
	FILE RECORD
	FILE RECORD KEYED
	FILE RECORD DIRECT KEYED UPDATE

F|FB|FS|FBS|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	N	—	S	Only F for REGIONAL(1) and (2)

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 INDEXED and REGIONAL output.

Figure 45 (Part 1 of 2). Attributes and Options of PL/I File Declarations

Types of File / File Attributes and ENVIRONMENT Options	Record: Sequential: Consecutive: Stream	Record: Sequential: Consecutive: Buffered	Record: Sequential: Consecutive: Unbuffered	Record: Sequential: Teleprocessing	Record: Sequential: Indexed	Record: Sequential: Regional: Buffered	Record: Sequential: Regional: Unbuffered	Record: Direct: VSAM	Record: Direct: VSAM	Record: Direct: Indexed	Record: Direct: Regional	Comments
F|FB|V|VB	–	–	–	–	S	–	–	N	N	S	–	VS invalid with UNBUF
RECSIZE(n)	I	I	I	S	I	I	I	C	C	I	I	One or both must be specified for consecutive, indexed, and regional files
BLKSIZE(n)	I	I	I	–	I	I	I	N	N	I	I	
NCP(n)	–	O	O	–	O	O	O	N	N	O	O	NCP>1 for VSAM specifies ISAM compatibility
TRKOFL	–	O	O	–	–	O	O	–	–	–	O	Invalid for REGIONAL(3)
KEYLENGTH(n)	–	–	–	–	S	S	S	C	C	S	S	For REGIONAL(2) and (3) OUTPUT only
COBOL	–	O	O	–	O	O	O	O	O	O	O	
BUFFERS(n)	I	I	–	I	I	I	–	N	N	–	–	
SCALARVARYING	–	O	O	–	O	O	O	O	O	O	O	Invalid for ASCII data sets
CONSECUTIVE	D	D	D	–	–	–	–	O	O	–	–	Allowed for VSAM ESDS
TOTAL	–	O	–	–	–	–	–	–	–	–	–	
LEAVE	O	O	O	–	–	–	–	–	–	–	–	
REREAD	O	O	O	–	–	–	–	–	–	–	–	
ASCII	O	O	–	–	–	–	–	–	–	–	–	
BUFOFF(n)	O	O	–	–	–	–	–	–	–	–	–	
CTLASA|CTL360	–	O	O	–	–	–	–	–	–	–	–	Invalid for ASCII data sets
GRAPHIC	O	–	–	–	–	–	–	–	–	–	–	
TP({M|R})	–	–	–	S	–	–	–	–	–	–	–	
INDEXED	–	–	–	–	S	–	–	O	O	S	–	Allowed for VSAM ESDS
KEYLOC(n)	–	–	–	–	O	–	–	–	–	O	–	
INDEXAREA(n)	–	–	–	–	–	–	–	–	–	O	–	
ADDBUFF	–	–	–	–	–	–	–	–	–	O	–	
NOWRITE	–	–	–	–	–	–	–	–	–	O	–	UPDATE files only
GENKEY	–	–	–	–	O	–	–	O	O	O	–	INPUT or UPDATE files only; KEYED is required
REGIONAL ({1|2|3})	–	–	–	–	–	S	S	–	–	–	S	
VSAM	–	–	–	–	–	–	–	S	S	–	–	
PASSWORD	–	–	–	–	–	–	–	O	O	–	–	
SIS	–	–	–	–	–	–	–	–	O	–	–	
SKIP	–	–	–	–	–	–	–	O	–	–	–	
BKWD	–	–	–	–	–	–	–	O	–	–	–	
REUSE	–	–	–	–	–	–	–	O	O	–	–	OUTPUT file only
BUFND(n)	–	–	–	–	–	–	–	O	O	–	–	
BUFNI(n)	–	–	–	–	–	–	–	O	O	–	–	
BUFSP(n)	–	–	–	–	–	–	–	O	O	–	–	

Key:

- I Must be specified or implied
- C Checked for VSAM
- D Default
- N Ignored for VSAM
- O Optional
- S Must be specified
- – Invalid

Figure 45 (Part 2 of 2). Attributes and Options of PL/I File Declarations

OTHER ENVIRONMENT OPTIONS

A constant or variable can be used with those ENVIRONMENT options that require integer arguments, such as block sizes and record lengths. The variable must be unsubscripted, unqualified, and have attributes FIXED BINARY(31,0) and STATIC.

Some of the information that can be specified in the options of the ENVIRONMENT attribute can also be specified, when TOTAL is not specified, in the subparameters of the DCB parameter of a DD statement. Figure 46 gives a list of equivalents.

ENVIRONMENT Option	DCB Subparameter
Record format	RECFM[1]
RECSIZE	LRECL
BLKSIZE	BLKSIZE
BUFFERS	BUFNO
CTLASA|CTL360	RECFM
NCP	NCP
TRKOFL	RECFM
KEYLENGTH	KEYLEN
KEYLOC	RKP
ASCII	ASCII
BUFOFF	BUFOFF

[1] VS must be specified as an ENVIRONMENT option, not in the DCB.

Figure 46. Equivalent ENVIRONMENT Options and DCB Subparameters

Record Format Options for Record-Oriented Data Transmission

Record formats supported depend on the data set organization.

Syntax

```
F|FB|FS|FBS|V|VB|VS|VBS|D|DB|U
```

Records can have one of the following formats:

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

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

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

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

Record Format Options for Stream-Oriented Data Transmission

The record format options for stream-oriented data transmission are discussed in Chapter 5, "Defining Data Sets for Stream Files" on page 134.

RECSIZE Option

The RECSIZE option specifies the record length.

Syntax

```
RECSIZE(record-length)
```

For files other than transient files and files associated with VSAM data sets, "record-length" is the sum of:

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

For a transient file, it is the sum of:

1. The 4 V-format control bytes
2. One flag byte
3. Eight bytes for the key (origin or destination identifier)
4. The maximum length required for the data

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 the RECSIZE option is included in the file declaration for checking purposes, the maximum record size should be specified. If RECSIZE is specified and conflicts with the values defined for the data set, the UNDEFINEDFILE condition is raised.

The "record-length" can be specified as an integer or as a variable with the attributes FIXED BINARY(31,0) STATIC.

The value is subject to the following conventions:

Maximum:

Fixed-length, and undefined (except ASCII data sets): 32,760 bytes

V-format, and VS- and VBS-format with UPDATE files: 32,756 bytes

VS- and VBS-format with INPUT and OUTPUT files: no limit

ASCII data sets: 9999

VSAM data sets: 32,761 for nonspanned records. For spanned records, the maximum is the size of the control area.

For VS- and VBS-format records longer than 32,756 bytes, the length must be specified in the RECSIZE option of ENVIRONMENT, and the DCB subparameter of the DD statement must specify LRECL=X.

Zero value:
A search for a valid value is made (in the following order) in the:

- DD statement for the data set associated with the file
- Data set label

If neither of these can provide a value, default action is taken (see "Record Format, BLKSIZE, and RECSIZE Defaults" on page 129).

Negative Value:
The UNDEFINEDFILE condition is raised.

BLKSIZE Option

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

Syntax

```
BLKSIZE(block-size)
```

The "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 nonspanned. 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.

   or

   Any block prefix bytes required (ASCII data sets only).

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

"block-size" is subject to the following conventions:

Maximum:
32,760 bytes (or 9999 for an ASCII data set for which BUFOFF without a prefix-length value has been specified).

In regional 3 files, the maximum DECLAREd blocksize must not exceed 32,680 bytes. This is because the 32,760 byte maximum for blocksize consists of the declared blocksize plus the key length plus the length of the IOCB. If 'BLKSIZE=32760' is declared, when the keylength and IOCB length are added to it, the maximum will be exceeded and an "UNDEFINED FILE" error message will be issued.

Zero value:
A search for a valid value is made (in the following order) in the:

- DD statement for the data set associated with the file
- Data set label

If neither of these can provide a value, default action is taken (see "Record Format, BLKSIZE, and RECSIZE Defaults" on page 129)

Negative value:
The UNDEFINEDFILE condition is raised

The relationship of the "block-size" to the "record-length" depends on the record format:

FB-format or FBS-format:
The block size must be a multiple of the record length

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

1. The maximum length of any record
2. Four control bytes

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

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:

- The BLKSIZE option can be used with unblocked (F-, V-, or D-format) records, as follows:
  - The BLKSIZE option, but not the RECSIZE option, is specified. The record length is set equal to the block size (minus any control or prefix bytes), and the record format is unchanged.
  - Both the BLKSIZE and the RECSIZE options are specified, and the relationship of the two values is compatible with blocking for the record format used. The record format is set to FB, VB, or DB, whichever is appropriate.
- If, for FB-format or FBS-format records, the block size equals the record length, the record format is set to F.
- 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 it is specified for one.

Record Format, BLKSIZE, and RECSIZE Defaults

If, for a non-VSAM data set, either the record-format, BLKSIZE, or RECSIZE option is not specified, the following action is taken:

Record format:
A search is made in the associated DD statement or data set label. If the search does not provide a value, the UNDEFINEDFILE condition is raised, except for files associated with dummy data sets or the foreground terminal, in which case the record format is set to U.

Block-size or record-length:
If one of these is specified, a search is made for the other in the associated DD statement or data set label. If the search provides a value, and if this value is incompatible with the value in the specified option, the UNDEFINEDFILE condition is raised. If the search is unsuccessful, a value is derived from the specified option (with the addition or subtraction of any control or prefix bytes). If neither is specified, the UNDEFINEDFILE condition is raised, except for files associated with dummy data sets, in which case a "block-size" is set to 121 for F-format or U-format records and to 129 for V-format records. For files associated with the foreground terminal, the "record-length" is set to 120.

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 the blocking and deblocking of records and for locate-mode transmission.

The option BUFFERS(n) in the ENVIRONMENT attribute specifies, for CONSECUTIVE and INDEXED data sets, the number, n, of buffers to be allocated for a data set; this number must not exceed 255 (or such other maximum as was established at system generation).

Syntax

```
BUFFERS(n)
```

If the number of buffers is not specified for a BUFFERED file or is specified as zero, two buffers are used by the optimizing compiler, and one buffer is used by the checkout compiler. A REGIONAL data set is always allocated two buffers.

In teleprocessing, the BUFFERS option specifies the number of buffers available for a particular message queue; that is, for a particular TRANSIENT file. The buffer size is specified in the message control program for the installation. The number of buffers specified should, if possible, be sufficient to provide for the longest message to be transmitted.

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

GENKEY Option—Key Classification

The GENKEY (generic key) option applies only to INDEXED and VSAM key-sequenced data sets. It enables you to classify keys recorded in a data set and to use a SEQUENTIAL KEYED INPUT or SEQUENTIAL KEYED UPDATE file to access records according to their key classes.

Syntax

GENKEY

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

The GENKEY option allows you to start sequential reading or updating of a VSAM data set from the first record that has a key in a particular class, and for an INDEXED data set from the first nondummy record that has a key in a particular class. The class is identified by the inclusion of its generic key in the KEY option of a READ statement. Subsequent records can be read by READ statements without the KEY option. No indication is given when the end of a key class is reached.

Although the first record having a key in a particular class can be retrieved by a READ with the KEY option, the actual key cannot be obtained unless the records have embedded keys, since the KEYTO option cannot be used in the same statement as the KEY option.

In the following example, a key length of more than 3 bytes is assumed:

```
DCL IND FILE RECORD SEQUENTIAL KEYED
   UPDATE ENV (INDEXED GENKEY);
          .
          .
          .
       READ FILE(IND) INTO(INFIELD)
                      KEY ('ABC');
          .
          .
          .
NEXT:  READ FILE (IND) INTO (INFIELD);
          .
          .
          .
       GO TO NEXT;
```

The first READ statement causes the first nondummy record in the data set whose key begins with 'ABC' to be read into INFIELD; each time the second READ statement is executed, the nondummy record with the next higher key is retrieved. Repeated execution of the second READ statement could result in reading records from higher key classes, since no indication is given when the end of a key class is reached. It is your responsibility to check each key if you do not wish to read beyond the key class. Any subsequent execution of the first READ statement would reposition the file to the first record of the key class 'ABC'.

If the data set contains no records with keys in the specified class, or if all the records with keys in the specified class are dummy records, the KEY condition is raised. The data set is then positioned either at the next record that has a higher key or at the end of the file.

Note how the presence or absence of the GENKEY option affects the execution of a READ statement that supplies a source key that is shorter than the key length specified in the KEYLEN subparameter of the DD statement that defines the indexed data set. GENKEY causes the key to be interpreted as a generic key, and the data set is positioned to the first nondummy record in the data set whose key begins with the source key. For a READ

statement, if the GENKEY option is not specified, a 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.

The 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).

NCP Option—Number of Channel Programs

The NCP option specifies the number of incomplete input/output operations with the EVENT option that can be handled for the file at any one time.

Syntax

```
NCP(n)
```

The integer, n, specified with NCP must have a value in the range 1 through 99; otherwise the default is 1.

For consecutive and regional sequential files, it is an error to allow more than the specified number of events to be outstanding.

For indexed files, any excess operations are queued, and no condition is raised. However, specification of the number of channel programs required may aid optimization of I/O with an indexed file. The NCP option has no effect with a regional direct file.

A file declared with ENVIRONMENT(VSAM) can never have more than one incomplete input/output operation at any one time. If the NCP option is specified for such a file, it is ignored. For information about the NCP option for VSAM with the ISAM compatibility interface, see "The VSAM Compatibility Interface" on page 234.

TRKOFL Option—Track Overflow

Track overflow is a feature of the operating system that can be incorporated at system generation time; it requires the record overflow feature on the direct-access storage control unit. Track overflow allows a record to overflow from one track to another. It is useful in achieving a greater data packing efficiency, and allows the size of a record to exceed the capacity of a track.

Syntax

```
TRKOFL
```

Track overflow is not available for REGIONAL(3) or INDEXED data sets.

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.

Syntax

```
COBOL
```

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

- A file with the COBOL option can be used only for READ INTO, WRITE FROM, and REWRITE FROM statements.
- The file name cannot be passed as an argument or assigned to a file variable.
- The variable to be transmitted must be subscripted.
- If a condition is raised during the execution of a READ statement, the variable named in the INTO option cannot be used in the on-unit. If the completed INTO variable is required, there must be a normal return from the on-unit.
- The EVENT option can be used only if the compiler can determine 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.

For supported COBOL compilers and for PL/I equivalents of COBOL data types, see Chapter 14, "Interlanguage Communication with COBOL and FORTRAN" on page 343.

SCALARVARYING Option—Varying-Length Strings

The SCALARVARYING option is used in the input/output of varying-length strings, and can be specified with records of any format.

Syntax

```
SCALARVARYING
```

When storage is allocated for a varying-length string, the compiler includes a 2-byte prefix that specifies the current length of the string. For an element varying-length string, this prefix is included on output, or recognized on input, only if SCALARVARYING is specified for the file.

When locate mode statements (LOCATE and READ SET) are used to create and read a data set with element varying-length strings, SCALARVARYING must be specified 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 SCALARVARYING is specified and element varying-length strings are transmitted, you must allow 2 bytes in the record length to include the length prefix.

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

SCALARVARYING and CTLASA/CTL360 must not be specified for the same file, as this causes the first data byte to be ambiguous.

KEYLENGTH Option

The KEYLENGTH option specifies the length, n, of the recorded key for KEYED files. KEYLENGTH can be specified for INDEXED or REGIONAL(3) files.

Syntax

```
KEYLENGTH(n)
```

If the KEYLENGTH option is included in a VSAM file declaration for checking purposes, and the key length specified in the option conflicts with the value defined for the data set, the UNDEFINEDFILE condition is raised.

CHAPTER 5. DEFINING DATA SETS FOR STREAM FILES

This chapter describes how to define data sets for use with PL/I files that have the STREAM attribute. It lists the ENVIRONMENT options that can be used and explains how to create and access data sets. The essential parameters of the DD statements used 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 the 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.

Data sets for stream-oriented data transmission are created and accessed using the list-, data-, and edit-directed input and output statements described in Chapter 13 of the OS and DOS PL/I Language Reference Manual.

For output, PL/I converts the data items from the 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 the program, converting them into the appropriate form for assignment to the program variables.

┌──────────── Optimizing Compiler Only ────────────┐

Stream-oriented data transmission can be used 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 device or for a print utility program such as the Kanji print utility.[3] For example, the Kanji print utility does not allow graphic strings to be continued onto another line.

└──────────── End of Optimizing Compiler Only ────────────┘

DEFINING FILES FOR STREAM-ORIENTED DATA TRANSMISSION

Files for stream-oriented data transmission are defined by a file declaration with the following attributes:

```
DCL filename FILE STREAM
             INPUT | {OUTPUT [PRINT]}
             ENVIRONMENT(options);
```

Default file attributes are shown in Figure 45 on page 123; the FILE attribute is described in the OS and DOS PL/I Language Reference Manual. The PRINT attribute is described further in "Print Files" on page 143. Options of the ENVIRONMENT attribute are discussed below.

---

[3] Details on processing Japanese or Chinese graphics are available through the IBM World Trade Americas/Far East Corporation.

ENVIRONMENT OPTIONS

The options applicable to stream-oriented data transmission are as follows. The options are described in this chapter, except for BLKSIZE and BUFFERS, which are described in "Data Set Organization Options" on page 122, and LEAVE, REREAD, ASCII, and BUFOFF, which are described under "Consecutive Data Sets" on page 149.

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

LEAVE
REREAD
ASCII
BUFOFF[(n)]
```

For more information, see Figure 45 on page 123.

CONSECUTIVE Option

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 "Consecutive Data Sets" on page 149.

Syntax

```
CONSECUTIVE
```

Record Format Options

Although record boundaries are ignored in stream-oriented data transmission, record format is important when a data set is being created, not only because it 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 may 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 as a series of characters or graphics arranged in lines, and 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.

Syntax

```
F|FB|FBS|FS|V|VB|D|DB|U
```

Records can have one of the following formats, as described in Chapter 4, "Data Sets and Files" on page 100.

Blocking and deblocking of records are performed automatically.

Fixed-length	F	unblocked
	FB	blocked
	FBS	blocked, standard
	FS	unblocked, standard
Variable-length	V	unblocked
	VB	blocked
	D	unblocked ASCII
	DB	blocked ASCII
Undefined-length	U	(cannot be blocked)

RECSIZE Option

RECSIZE for stream-oriented data transmission is the same as that described in "Data Set Organization Options" on page 122. 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 OS and DOS PL/I Language Reference Manual.

Additional record-size considerations for list- and data-directed transmission of graphics are given in Chapter 13 of the OS and DOS PL/I Language Reference Manual.

Record Format, BLKSIZE, and RECSIZE Defaults

If the record format, BLKSIZE, or RECSIZE option is not specified in the ENVIRONMENT attribute, or in the associated DD statement or data set label, the following action is taken:

INPUT files:

Defaults are applied as for record-oriented data transmission, described under "Record Format, BLKSIZE, and RECSIZE Defaults" on page 129.

Output Files:

Record format:
Set to VB-format, or if ASCII option specified, to DB-format

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

PRINT files:

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

Non-PRINT files:

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

Block size:

F, FB, or FBS:	record length
V or VB:	record length + 4
D or DB:	record length + block prefix (see note)

Optimizing Compiler Only

GRAPHIC Option

The GRAPHIC option of the ENVIRONMENT attribute must be specified if you use graphic variables or graphic constants in GET and PUT statements for list- and data-directed input/output, and can be specified for edit-directed input/output.

Syntax

```
GRAPHIC
```

For list- and data-directed input/output, if you have graphics in input or output data and do not specify the GRAPHIC option, the ERROR condition is raised.

For edit-directed input/output, the GRAPHIC option specifies that left and right delimiters are to be added to graphic

variables and constants on output, and that input graphics will have left and right delimiters. If the GRAPHIC option is not specified, left and right delimiters will not be added to output data, and input graphics do not require left and right delimiters. When the GRAPHIC option is specified, 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 input/output, see the OS and DOS PL/I Language Reference Manual.

└──────── End of Optimizing Compiler Only ────────┘

CREATING A DATA SET FOR STREAM-ORIENTED DATA TRANSMISSION

To create a data set, you must give the operating system certain information either in your PL/I program or in the DD statement that defines the data set. The following paragraphs indicate the essential information, and discuss some of the optional information you may supply.

ESSENTIAL INFORMATION

You must supply the following information, summarized in Figure 47 on page 138, when creating a data set:

- Device that will write or punch your data set (UNIT, SYSOUT, or VOLUME parameter of DD statement).
- Block size: You can specify the block size either in your PL/I program (ENVIRONMENT attribute or LINESIZE option of the OPEN statement) or in the DD statement (BLKSIZE subparameter). If you do not specify a record length, unblocked records are the default and the record length is determined from the block size. If you do not specify a record format, U-format is the default (except for PRINT files when V-format is the default; see "Print Files" on page 143).

If you want to keep a magnetic-tape or direct-access data set (that is, you do not want the operating system to delete it at the end of your job), the DD statement must name the data set and indicate how it is to be disposed of (DSNAME and DISP parameters). The DISP parameter alone will suffice if you want to use the data set in a later step but will not need it after the end of your job.

When creating a data set on a direct-access device, you must specify the amount of space required for it (SPACE parameter of DD statement).

If you want your data set stored on a particular magnetic-tape or direct-access device, you must indicate the volume serial number in the DD statement (SER or REF subparameter of VOLUME parameter). If you do not supply a serial number for a magnetic-tape data set that you want to keep, the operating system will allocate one, inform the operator, and print the number on your program listing.

If your data set is to follow another data set on a magnetic-tape volume, you must use the LABEL parameter of the DD statement to indicate its sequence number on the tape.

Parameters of DD Statement

Storage Device	When Required	What You Must State	Parameters
All	Always	Output device	UNIT= or SYSOUT= or VOLUME=REF=
		Block size[1]	DCB=(BLKSIZE=...)
Direct access only	Always	Storage space required	SPACE=
Magnetic tape only	Data set not first in volume and for magnetic tapes that do not have standard labels	Sequence number	LABEL=
Direct access and standard labeled magnetic tape	Data set to be used by another job step but is not required after end of job	Disposition	DISP=
	Data set to be kept after end of job	Disposition	DISP=
		Name of data set	DSNAME=
	Data set to be on particular volume	Volume serial number	VOLUME=SER or VOLUME=REF=

[1]Alternatively, you can specify the block size in your PL/I program by using either the ENVIRONMENT attribute or the LINESIZE option.

Figure 47. Creating a Data Set for Stream-Oriented Data Transmission: Essential Parameters of DD Statement

EXAMPLES

The use of edit-directed stream-oriented data transmission to create a data set on an IBM 3330 Disk Storage is shown in Figure 48 on page 139. The data read from the input stream by the file SYSIN includes a field VREC that contains five unnamed 7-character subfields; the field NUM defines the number of these subfields that contain information. The output file WORK transmits to the data set the whole of the field FREC and only those subfields of VREC that contain information.

```
//EX7#2 JOB
//STEP1 EXEC PLIXCLG
//PLI.SYSIN  DD  *
 PEOPLE: PROC OPTIONS(MAIN);
         DCL WORK FILE STREAM OUTPUT,
             1 REC,
               2 FREC,
                 3 NAME CHAR(19),
                 3 NUM CHAR(1),
                 3 PAD CHAR(25),
               2 VREC CHAR(35),
             IN CHAR(80) DEF REC;
         ON ENDFILE(SYSIN) GO TO FINISH;
         OPEN FILE(WORK) LINESIZE(400);
 MORE:   GET FILE(SYSIN) EDIT(IN)(A(80));
         PUT FILE(WORK) EDIT(IN)(A(45+7*NUM));
         GO TO MORE;
 FINISH: CLOSE FILE(WORK);
         END PEOPLE;
/*
//GO.WORK DD DSN=HPU8.PEOPLE,UNIT=SYSDA,SPACE(TRK,(1,1)),
//            DISP=(NEW,CATLG),VOL=SER=nnnnnn
//GO.SYSIN DD *
R.C.ANDERSON        0 202848 DOCTOR
B.F.BENNETT         2 771239 PLUMBER          VICTOR HAZEL
R.E.COLE            5 698635 COOK             ELLEN  VICTOR JOAN   ANN    OTTO
J.F.COOPER          5 418915 LAWYER           FRANK  CAROL  DONALD NORMAN BRENDA
A.J.CORNELL         3 237837 BARBER           ALBERT ERIC   JANET
E.F.FERRIS          4 158636 CARPENTER        GERALD ANNA   MARY   HAROLD
/*
```

Figure 48. Creating a Data Set with Stream-Oriented Data Transmission

Figure 49 on page 140 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.

```
XAMPLE1:  PROC OPTIONS(MAIN);
          DCL    INFILE     FILE    INPUT     RECORD,
                 OUTFILE    FILE    OUTPUT    STREAM ENV(GRAPHIC);
                                              /* GRAPHIC OPTION MEANS */
          DCL                                 /* DELIMITERS WILL BE   */
                 1 IN,                        /* INSERTED ON OUTPUT   */
                   3  EMPNO  CHAR(6),         /* FILES.               */
                   3  NAME,
                      5  LAST   G(7),         /* THIS DATA REQUIRES   */
                      5  FIRST  G(7),         /* SPECIAL INPUT DEVICE */
                   3  ADDRESS,                /* TO INPUT GRAPHIC     */
                      5  ZIP    CHAR(6),      /* CHARACTER.           */
                      5  DISTRICT G(5),
                      5  CITY     G(5),
                      5  OTHER    G(10);
          DCL    ADDRWK         G(22);
          ON     ENDFILE(INFILE) GO TO LAST;
READ:
          READ FILE(INFILE) INTO(IN);
          IF   SUBSTR(ZIP,1,3)-='300'
               THEN GO TO READ;
          L=0;
          ADDRWK=DISTRICT;                    /* ASSIGNMENT STATEMENT */
          DO I=1 TO 5;

          IF SUBSTR(DISTRICT,I,1)= ├' ' ' ' G┤   /* SUBSTR BIF PICKS UP  */

             THEN GO TO NEXT1;                    /* THE ITH GRAPHIC CHAR */
          END;                                    /* IN DISTRICT.         */
NEXT1:    L=L+I+1;
          SUBSTR(ADDRWK,L,5)=CITY;
          DO I=1 TO 5;

          IF SUBSTR(CITY,I,1)= ├' ' ' ' G┤


             THEN GO TO NEXT2;
          END;
NEXT2:    L=L+I;
          SUBSTR(ADDRWK,L,10)=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(22)); /* TO PRINT GRAPHIC */
            GO TO READ;                     /* DATA.             */
LAST:
          END XAMPLE1;
```

Figure 49. Writing Graphic Data to a Stream File

ACCESSING A DATA SET FOR STREAM-ORIENTED DATA TRANSMISSION

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 a DD statement. The following paragraphs, which are summarized in Figure 50 on page 141, indicate the essential information you must include in the DD statement, and discuss some of the optional information you may supply. The discussions do not apply to data sets in the input stream.

ESSENTIAL INFORMATION

If the data set is cataloged, you need supply only the following information in the DD statement:

- The name of the data set (DSNAME parameter). The operating system will locate the information describing the data set in the system catalog, and, if necessary, will request the operator to mount the volume containing it.
- Confirmation that the data set exists (DISP parameter). If you open the data set for output with the intention of extending it by adding records at the end, code DISP=MOD; otherwise, opening the data set for output will result in it being overwritten.

If the data set is not cataloged, you must, in addition, specify the device that will read the data set and, for magnetic-tape and direct-access devices, give the serial number of the volume that contains the data set (UNIT and VOLUME parameters).

If the data set is on paper tape or punched cards, you must specify the block size either in your PL/I program (ENVIRONMENT attribute) or in the DD statement (BLKSIZE subparameter).

If the data set follows another data set on a magnetic-tape volume, you must use the LABEL parameter of the DD statement to indicate its sequence number on the tape.

Parameters of DD Statement

When Required		What You Must State	Parameters
Always		Name of data set	DSNAME=
		Disposition of data set	DISP=
If data set not cataloged	All devices	Input device	UNIT= or VOLUME=REF=
	Standard labeled magnetic tape and direct access	Volume serial number	VOLUME=SER=
Magnetic tape: if data set not first in volume or which does not have standard labels		Sequence number	LABEL=
If data set does not have standard labels		Block size[1]	DCB=(BLKSIZE=...)

[1] Alternatively, you can specify the block size in your PL/I program by using either the ENVIRONMENT attribute or the LINESIZE option.

Figure 50. Accessing a Data Set: Essential Parameters of DD Statement

Magnetic Tape Without IBM Standard Labels

If a magnetic-tape data set has nonstandard labels or is unlabeled, you must specify the block size either in your PL/I program (ENVIRONMENT attribute) or in the DD statement (BLKSIZE subparameter). The DSNAME parameter is not essential if the data set is not cataloged.

PL/I includes no facilities for processing nonstandard labels, which, to the operating system, appear as data sets preceding or following your data set. You can either process the labels as independent data sets or use the LABEL parameter of the DD statement to bypass them. To bypass the labels, code LABEL=(2,NL) or LABEL=(,BLP).

```
//EX7#5 JOB
//STEP1 EXEC PLIXCLG
//PLI.SYSIN  DD  *
 PEOPLE: PROC OPTIONS(MAIN);
         DCL WORK FILE STREAM INPUT,
             1 REC,
               2 FREC,
                 3 NAME CHAR(19),
                 3 NUM CHAR(1),
                 3 SERNO CHAR(7),
                 3 PROF CHAR(18),
               2 VREC CHAR(35),
             IN CHAR(80) DEF REC;
         ON ENDFILE(WORK) GO TO FINISH;
         OPEN FILE(WORK);
 MORE:   GET FILE(WORK) EDIT(IN,VREC)(A(45),A(7*NUM));
         PUT FILE(SYSPRINT) SKIP EDIT(IN)(A);
         GO TO MORE;
 FINISH: CLOSE FILE(WORK);
         END PEOPLE;
/*
//GO.WORK DD DSN=HPU8.PEOPLE,DISP=(OLD,DELETE)
```

Figure 51. Accessing a Data Set with Stream-Oriented Data Transmission

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 51 on page 142 reads the data set created by the program in Figure 48 on page 139 and uses the file SYSPRINT to list the data it contains. (For details on SYPRINT, see "SYSIN and SYSPRINT Files" on page 147.) Each set of data is read, by the GET statement, into two variables: FREC, which always contains 45 characters; and VREC, which always contains 35 characters. At each execution of the GET statement, VREC consists of the number of characters generated by the expression 7*NUM, together with sufficient blanks to bring the total number of characters to 35. The DISP parameter of the DD statement could read simply DISP=OLD; if DELETE is omitted, an existing data set will not be deleted.

PRINT FILES

Both the operating system and the PL/I language include features that facilitate the formatting of printed output. The operating system allows you to use the first byte of each record for a print control character; the control characters, which are not printed, cause the printer to skip to a new line or page. Tables of print control characters are given in Figure 62 on page 163 and Figure 63 on page 164. 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
b (blank)	Space 1 line before printing
0	Space 2 lines before printing
-	Space 3 lines before printing
+	No space before printing
1	Start new page

The compiler handles the PAGE, SKIP, and LINE options or format items by padding the remainder of the current record with blanks and inserting the appropriate control character in the next record. If SKIP or LINE specifies more than a 3-line space, the compiler inserts sufficient blank records with appropriate control characters to accomplish the required spacing. In the absence of a print control option or format item, when a record is full the compiler inserts a blank character (single line space) in the first byte of the next record.

If a PRINT file is being transmitted to a terminal, the PAGE, SKIP, and LINE options will never cause more than 3 lines to be skipped, unless formatted output is specified (see the CMS User's Guide or TSO User's Guide).

RECORD FORMAT

You can limit the length of the printed line produced by a PRINT file either by specifying a record length in your PL/I program (ENVIRONMENT attribute) or in a DD statement, or by giving a line size in an OPEN statement (LINESIZE option). The record length must include the extra byte for the print control character, that is, it must be 1 byte larger than the length of the printed line (5 bytes larger for V-format records). The value you specify in the LINESIZE option refers to the number of characters in the printed line; the compiler adds the print control character.

The blocking of records has no effect on the appearance of the output produced by a PRINT file, but it does result in more efficient use of auxiliary storage when the file is associated with a data set on a 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.

Although you can vary the line size for a PRINT file during execution by closing the file and opening it again with a new line size, you must do so with caution if you are using the PRINT file to create a data set on a magnetic-tape or direct-access device; you cannot change the record format established for the data set when the file is first opened. If the line size specified in an OPEN statement conflicts with the record format already established, the UNDEFINEDFILE condition will be raised; to prevent this, either specify V-format records with a block size at least 9 bytes greater than the maximum line size you intend to use, or ensure that the first OPEN statement specifies the maximum line size. (Output destined for the printer may be stored temporarily on a direct-access device, unless you specify a printer by using UNIT=, even if you intend it to be fed directly to the printer.)

Since PRINT files have a default line size of 120 characters, you need not give any record format information for them. In the absence of other information, the compiler assumes V-format records; the complete default information is:

```
BLKSIZE=129

LRECL=125

RECFM=VBA
```

EXAMPLE

Figure 52 on page 145 illustrates the use of a PRINT file and the printing options of stream-oriented data transmission statements to format a table and write it onto a direct-access device for printing on a later occasion. The table comprises the natural sines of the angles from 0° to 359° 54' in steps of 6'.

The statements in the ENDPAGE on-unit insert a page number at the bottom of each page, and set up the headings for the following page.

The DD statement defining the data set created by this program includes no record-format information; the compiler infers the following from the file declaration and the line size specified in the statement that opens the file TABLE:

```
Record format =
          V (the default for a PRINT file).
```

Record size =
98 (line size + 1 byte for print control character + 4 bytes for record control field).

Block size =
102 (record length + 4 bytes for block control field).

The program in Figure 67 on page 166 uses record-oriented data transmission to print the table created by the program in Figure 52.

```
//OPT7#5 JOB
//STEP1 EXEC PLIXCLG
//PLI.SYSIN  DD  *
 SINE: PROC OPTIONS(MAIN);

      DCL TABLE FILE STREAM OUTPUT PRINT,
          DEG FIXED DEC(5,1) INIT(0), /* INIT(0) FOR TEST IN ENDPAGE */
          MIN FIXED DEC(3,1),         /* INCREMENTS TO 1.0 IN DO-LOOP*/
          PGNO FIXED DEC(2) INIT(0);

      ON ENDPAGE(TABLE) BEGIN;
         DCL I;
         IF PGNO ¬= 0 THEN            /* ? FOOTING   */
            PUT FILE(TABLE) EDIT
             ('PAGE',PGNO)       (LINE(55),COL(80),A,F(3));
         IF DEG ¬= 360 THEN           /* ? HEADING   */
            DO;
            PUT FILE(TABLE) PAGE EDIT
                ('NATURAL SINES') (A);
            IF PGNO ¬= 0 THEN         /* ? HEADING CONTINUED */
               PUT FILE(TABLE) EDIT
                (' (CONT''D)')   (A);
            PUT FILE(TABLE) EDIT
                ((I DO I = 0 TO 54 BY 6)) (SKIP(3),10 F(9));
            PGNO = PGNO +1;
            END;
            ELSE PUT FILE(TABLE) PAGE;
         END;

      OPEN FILE(TABLE) PAGESIZE(52) LINESIZE(93);
      SIGNAL ENDPAGE(TABLE);          /* HEADING - FIRST PAGE   */

      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);       /* FORCE LAST PAGE FOOTING
                                         (SIGNAL ENDPAGE CANNOT BE USED,
                                          WHEN PRINTING FOOTING PAST
                                          PAGESIZE-SEE ENDPAGE COND.) */
      END SINE;
/*
//GO.TABLE DD DSN=HPU8.SINES,DISP=(NEW,CATLG,DELETE),
//            UNIT=SYSDA,SPACE=(TRK,(1,1)),VOL=SER=nnnnnn
```

Figure 52. Creating a Data Set Using a PRINT File

TAB CONTROL TABLE

Data-directed and list-directed output to a PRINT file are aligned on preset tabulator positions. The preset tab positions are given in the OS and DOS PL/I Language Reference Manual. The tab settings are stored in a table in the transient library module, IBMBSTAB. The definitions of the fields in the table are as follows:

OFFSET OF TAB COUNT:
Halfword binary integer that gives the offset of "Tab count," the field that indicates the number of tabs to be used.

PAGESIZE:
Halfword binary integer that defines the default page size. This page size is used for dump output to the PLIDUMP data set as well as for stream output.

LINESIZE:
Halfword binary integer that defines the default line size.

PAGELENGTH:
Halfword binary integer that defines the default page length for printing at a terminal. For use with TSO and CMS. The value 0 indicates unformatted output.

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-Tabn:
n halfword binary integers that defines 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.

The preset PL/I tab settings can be overridden for your program by causing the linkage editor to resolve an external reference to PLITABS. To cause the reference to be resolved, supply a table with the name PLITABS, in the format described above.

There are two methods of supplying the tab table. One method is to include a PL/I structure in your source program with the name PLITABS, which must be declared STATIC EXTERNAL. An example of the PL/I structure is shown in Figure 53 on page 147. This example creates three tab settings, in positions 30, 60, and 90, and uses the defaults for page size and line size. Note that TAB1 identifies the position of the second item printed on a line; the first item on a line always starts at the left margin. The first item in the structure is the offset to the NO_OF_TABS field; FILL1, FILL2, and FILL3 can be omitted by adjusting the offset value by -6.

```
DCL 1 PLITABS STATIC EXT,
    2 (OFFSET INIT(14),
      PAGESIZE INIT(60),
      LINESIZE INIT(120),
      PAGELENGTH INIT(0),
      FILL1 INIT(0),
      FILL2 INIT(0),
      FILL3 INIT(0),
      NO_OF_TABS INIT(3),
      TAB1 INIT(30),
      TAB2 INIT(60),
      TAB3 INIT(90)) FIXED BIN(15,0);
```

Figure 53. PL/I Structure PLITABS for Modifying the Preset Tab Settings

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

SYSIN AND SYSPRINT FILES

If your program includes a GET statement that does not include the FILE option, the compiler inserts the file name SYSIN; if it includes a PUT statement without the FILE option, the compiler inserts the name SYSPRINT.

┌──────── Optimizing Compiler Only ────────┐

If you do not declare SYSPRINT, the compiler gives the file the attribute PRINT in addition to the normal default attributes; the complete set of attributes will be:

```
FILE STREAM OUTPUT PRINT EXTERNAL
```

Since SYSPRINT is a PRINT file, the compiler also supplies a default line size of 120 characters and a V-format record. You need give only a minimum of information in the corresponding DD statement; if your installation uses the usual convention that the system output device of class A is a printer, the following is sufficient:

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

You can override the attributes given to SYSPRINT by the compiler by explicitly declaring or opening the file. If you do so, bear in mind that this file is also used by the error-handling routines of the compiler, and that any change you make in the format of the output from SYSPRINT will also apply to the format of execution-time error messages. When an error message is printed, eight blanks are inserted at the start of each line except the first. If you specify a line size of less than 72 characters, the messages will not be output to SYSPRINT.

└──────── End of Optimizing Compiler Only ────────┘

┌──────── Checkout Compiler Only ────────┐

The SYSPRINT file is always required by the compiler, so is defined in the compiler program. There is no need for you to declare it in the PL/I program, because you cannot open or close the SYSPRINT file in your program; this is done by the compiler.

A DD statement defining the data set for SYSPRINT should be supplied. SYSPRINT must have VBA record format, and the record size (LRECL) must be in the range 125 through 137 (corresponding to line sizes 120 through 132). LRECL can be specified on a DD statement in the usual way, and will be accepted by the compiler

provided it is within these limits. If you wish to change the line size of SYSPRINT during your PL/I program, you can close the file and then open it with the LINESIZE option. The line size, however, cannot be increased beyond the value implied by LRECL.

If an OPEN statement attempts to exceed this value, the line size will not be changed. The compiler sets the default for LRECL to 125 and then opens the file. Therefore, if you wish to use a line size greater than 120, the DD statement for SYSPRINT must contain DCB information defining LRECL as the largest line size required plus 5.

The default line size value for SYSPRINT is chosen as follows:

- If LRECL is specified on the DD statement, then line size equals LRECL-5 (regardless of the PLITABS value, if any).
- If no LRECL is specified, then the default for LRECL is set to 125 and the line size for SYSPRINT is the smaller of 120 and the value in PLITABS.

└──────── End of Checkout Compiler Only ────────┘

If you use one of the IBM-supplied cataloged procedures to execute your program, the SYSPRINT DD statement is not required, since it is included in the GO procedure step.

The compiler does not supply any special attributes for the input file SYSIN; if you do not declare it, it receives only the default attributes. The data set associated with SYSIN is usually in the input stream; if it is not in the input stream, you must supply full DD information.

CHAPTER 6. USING CONSECUTIVE, INDEXED, REGIONAL, AND TELEPROCESSING DATA SETS

This chapter describes how to use consecutive, indexed, and regional data sets using the SAM, QSAM, ISAM and DAM access methods, and how to use teleprocessing data sets.

Figure 54 shows the facilities that are available with the various types of data sets that can be used with PL/I.

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.

CONSECUTIVE DATA SETS

This section describes consecutive data set organization and the ENVIRONMENT options that define consecutive data sets. It then describes how to create, access, and update consecutive data sets.

	VSAM KSDS	VSAM ESDS	VSAM RRDS	INDEXED	CONSECUTIVE	REGIONAL (1)	REGIONAL (2)	REGIONAL (3)
SEQUENCE	Key Order	Entry Order	Num-bered	Key Order	Entry Order	By Region	By Region	By Region
DEVICES	DASD	DASD	DASD	DASD	DASD, tape, card, etc.	DASD	DASD	DASD
ACCESS 1 By Key 2 Sequential 3 Backward	123	123	123	12	2 3 tape only	12	12	12
Alternate index Access as above	123	123	No	No	No	No	No	No
How Extended	With new keys	At end	In empty slots	With new keys	At end	In empty slots	With new keys	With new keys
SPANNED RECORDS	Yes	Yes	No	Yes	Yes	No	No	Yes
DELETION 1 Space reusable 2 Space not reusable	Yes, 1	No	Yes, 1	Yes, 2	No	Yes, 2	Yes, 2	Yes, 2

Figure 54. A Comparison of Data Set Types Available to PL/I Record I/O

CONSECUTIVE ORGANIZATION

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. The records can be retrieved only in the order in which they were written or in the reverse order when using the BACKWARDS attribute. The associated file must have the SEQUENTIAL attribute.

Figure 55 lists the data transmission statements and options that you can use to create and access a consecutive data set.

File declaration[1]	Valid statements[2] with options that must appear	Other options that can also be used
SEQUENTIAL OUTPUT BUFFERED	WRITE FILE(file-reference) FROM(reference); LOCATE based-variable FILE(file-reference);	 SET(pointer-reference
SEQUENTIAL OUTPUT UNBUFFERED	WRITE FILE(file-reference) FROM(reference);	EVENT(event-reference)
SEQUENTIAL INPUT BUFFERED[3]	READ FILE(file-reference) INTRO(reference); READ FILE(file-reference) SET(pointer-reference); READ FILE(file-reference) IGNORE(expression);	
SEQUENTIAL INPUT UNBUFFERED[3]	READ FILE(file-reference) INPUT(reference); READ FILE(file-reference) IGNORE(expression);	EVENT(event-reference) 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); READ FILE(file-reference) IGNORE(expression); REWRITE FILE(file-reference) FROM(reference);	EVENT(event-reference) EVENT(event-reference) EVENT(event-reference)

Figure 55. Statements and Options Permitted for Creating and Accessing Consecutive Data Sets

Notes to Figure 55:

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] The BACKWARDS attribute may be specified for files on magnetic tape.

DEFINING A CONSECUTIVE DATA SET

A consecutive data set is defined by 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 Figure 45 on page 123. The file attributes are described in the OS and DOS PL/I Language Reference Manual. Options of the ENVIRONMENT attribute are discussed below.

ENVIRONMENT OPTIONS FOR CONSECUTIVE DATA SETS

The ENVIRONMENT options applicable to consecutive data sets are:

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

CONSECUTIVE
TOTAL
LEAVE or REREAD
ASCII
BUFOFF[(n)]
CTLASA or CTL360
```

The options above the blank line are described in "Data Set Organization Options" on page 122, and those below the blank line are described below. D- and DB-format records are also described below.

See Figure 45 on page 123 to find which options must be specified, which are optional, and which are defaults.

CONSECUTIVE Option

The CONSECUTIVE option may be specified for a STREAM or RECORD file. It defines a file with consecutive data set organization, which is described above.

Syntax

```
CONSECUTIVE
```

CONSECUTIVE is the default when the merged attributes from the DECLARE and OPEN statements do not include the TRANSIENT attribute.

TOTAL Option — In-Line Code Optimization

In general, I/O operations are performed by library subroutines called from compiled code. Under certain conditions, however, the Optimizing Compiler can, when requested, provide in-line code to carry out these operations, thus saving the overhead of library calls. This gives faster execution of the I/O statements.

The TOTAL option aids the Optimizing Compiler in the production of efficient compiled 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 DCB parameter; 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.

Syntax

```
TOTAL
```

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 Checkout Compiler accepts and checks the TOTAL option but does not perform any optimization.

The TOTAL option cannot be specified for device-associated files or files reading Optical Mark Read data.

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

There are some differences in the optimized code that is generated under Release 5. The Release 5 implementation generates code to call modules in the PL/I Transient library so that mode-switching can be performed if necessary. This implementation results in a longer instruction path than on prior releases, but is still faster than not using the TOTAL option.

Figure 56 on page 153 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.

Statement[1]	Record Variable Requirements	File Attribute[3] or ENVIRONMENT Option Requirements[4]
READ SET	None	Not BACKWARDS for record types U, V, VB
READ INTO	Length known at compile time, maximum length for a varying string or area[2]	RECSIZE known at compile time.[5] SCALARVARYING option if varying string
WRITE FROM (fixed string)	Length known at compile time	RECSIZE known at compile time[5]
WRITE FROM (varying string)		RECSIZE known at compile time.[5] SCALARVARYING option used
WRITE FROM Area[2]		RECSIZE known at compile time[5]
LOCATE A	Length known at compile time, maximum length for a varying string or area[2]	RECSIZE known at compile time.[5] SCALARVARYING if varying string

Figure 56. Conditions Under Which I/O Statements Are Handled In-Line (TOTAL Option Used)

Notes to Figure 56:

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 File attributes are SEQUENTIAL BUFFERED, INPUT or OUTPUT.

4 Data set organization must be CONSECUTIVE; allowable record formats are F, FB, FS, FBS, U, V, or VB.

5 BLKSIZE may be specified instead of RECSIZE for unblocked record formats F, FS, V and U.

CTLASA and CTL360 Options - Printer and Punch Control

The printer/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.

Syntax

```
CTLASA | CTL360
```

The CTLASA option specifies American National Standard Vertical Carriage Positioning Characters or American National Standard Pocket Select Characters (Level 1). The CTL360 option specifies IBM machine-code control characters.

The control characters that can be used with these options are listed with their actions in "Punching Cards and Printing" on page 163.

LEAVE and REREAD Options - Magnetic Tape Handling

The magnetic tape handling options allow you to specify the action to be taken when the end of a magnetic tape volume is reached, or when a data set on a magnetic tape volume is closed. The LEAVE option prevents the tape from being rewound. The REREAD option rewinds the tape to permit reprocessing of the data set. If neither of these is specified, the action at end of volume or on closing of a data set is controlled by the DISP parameter of the associated DD statement.

Syntax

```
LEAVE | REREAD
```

If a data set is first read or written forward and then read backward in the same program, specify the LEAVE option to prevent rewind when the file is closed (or, with a multivolume data set, when volume switching occurs).

LEAVE and REREAD can also be specified on the CLOSE statement, as described in the OS and DOS PL/I Language Reference Manual.

The effects of the LEAVE and REREAD options are summarized in Figure 57 on page 155.

ENVIRONMENT Option	DISP Parameter	Action
REREAD	-	Positions the current volume to reprocess the data set. Repositioning for a BACKWARDS file is at the physical end of the data set.
LEAVE	-	Positions the current volume at the logical end of the data set. Repositioning for a BACKWARDS file is at the physical beginning of the data set.
Neither REREAD nor LEAVE	PASS	Positions the volume at the end of the data set
	DELETE	Rewinds the current volume
	KEEP, CATLG, UNCATLG	Rewinds and unloads the current volume

Figure 57. Effect of LEAVE and REREAD options

ASCII Option

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

Syntax

```
ASCII
```

Data sets on magnetic tape using ASCII may be created and accessed 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 with other data sets; D and DB formats, which correspond to V and VB formats with other data sets, are described below.

Only character data may be written onto an ASCII data set; when the data set is created, transmission must be from a character variable. This variable may have the attribute VARYING as well as CHARACTER, but the 2 length bytes of a varying-length character string cannot be transmitted; in other words, varying-length character strings cannot be transmitted to an ASCII data set using a SCALARVARYING file. Also, data aggregates containing varying-length strings may not be transmitted.

Since an ASCII data set must be on magnetic tape, it must be of consecutive organization. The associated file must be BUFFERED. The BUFOFF ENVIRONMENT option may be specified for ASCII data sets.

If ASCII is not specified in either the ENVIRONMENT option or the DD statement, but one of BUFOFF, D, or DB is specified, then ASCII is the default.

BUFOFF Option and Block Prefix Fields

At the beginning of each block in an ASCII data set, there may be a field known as the block prefix field. It may be from 1 to 99 bytes long. The buffer offset option, BUFOFF, specifies the length of this field to data management, so that the accessing or creation of data is started at this offset from the beginning of each physical block. PL/I does not support access to this field, and in general it does not contain information that is used in these implementations. There is one situation in which

data management does use information in the block prefix: with variable-length records (that is, D- or DB-format records), the block prefix field may be used to record the length of the block. In this case, it is 4 bytes long and contains a right-aligned, decimal character value that gives the length of the block in bytes, including the block prefix field itself. It is then exactly equivalent to a block length field.

Syntax

```
BUFOFF[(n)]
```

A numeric value equal to the length of the prefix may be specified for "n". It may be specified as either an integer or as a variable with the attributes FIXED BINARY(31,0) STATIC. Its minimum value is 0 and its maximum is 99. The absence of a prefix length specification indicates that the block prefix is to be used as a block length field; it implies that the field is 4 bytes long. The length of the block is inserted in the prefix by data management.

On input, any ASCII data set may be accessed if it has a block prefix field of length 1 to 99 bytes, or no block prefix field at all; and it may be accessed whether or not the block prefix field is used as a block length field.

On output, a data set using any one of the valid record formats may be created without a block prefix, but the only situation in which the creation of a block prefix is supported by PL/I is when it is used as a block length field. The only permissible buffer offset specification on output is therefore BUFOFF, with no prefix length specification.

The BUFOFF option may be used with ASCII data sets only.

BUFOFF Defaults

For output files, if you do not specify BUFOFF, the default is:

BUFFER offset:
F, FB, or U: 0
D, or DB: 4

With DB-format records on output files, the length of the block prefix (that is, the buffer offset) must always be either 0 or 4.

D-format and DB-format Records

The data contained in D- and DB-format records is recorded in ASCII. Each record may 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 4 control bytes contain the length of the record; this value is inserted by data management and requires no action by you. In addition, there may be, at the start of the block, a block prefix field, which may contain the length of the block.

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

Parameters of DD Statement			
Storage Device	**When Required**	**What You Must State**	**Parameters**
All	Always	Output device	UNIT= or SYSOUT= or VOLUME=REF=
		Block size[1]	DCB=(BLKSIZE=...
Direct access only	Always	Storage space required	SPACE=
Magnetic tape only	Data set not first in volume and for magnetic tapes that do not have standard labels	Sequence number	LABEL=
Direct access and standard labeled magnetic tape	Data set to be used by another job step but not required at end of job	Disposition	DISP=
	Data set to be kept after end of job	Disposition	DISP=
		Name of data set	DSNAME=
	Data set to be on particular device	Volume serial number	VOLUME=SER= or VOLUME=REF=

[1] Alternatively, you can specify the block size in your PL/I program by using the ENVIRONMENT attribute.

Figure 58. Creating a Consecutive Data Set: Essential Parameters of DD Statement

CREATING A CONSECUTIVE DATA SET

When you create a consecutive data set, the associated file must be opened for SEQUENTIAL OUTPUT. Either the WRITE or LOCATE statement may be used to write records. Figure 55 on page 150 shows the statements and options for creating a consecutive data set.

When creating a data set, you must identify it to the operating system in a DD statement. The following paragraphs, summarized in Figure 58, tell what essential information you must include in the DD statement and discuss some of the optional information you may supply.

Essential Information

When you create a consecutive data set you must specify the:

- Device that will write or punch your data set (UNIT, SYSOUT, or VOLUME parameter of DD statement). A data set with consecutive organization can exist on any type of auxiliary storage device.
- Block size: you can specify the block size either in your PL/I program (ENVIRONMENT attribute) or in the DD statement (BLKSIZE subparameter). If you do not specify a record length, unblocked records are the default and the record length is determined from the block size. If you do not specify a record format, U-format is the default.

If you want to keep a magnetic-tape or direct-access data set (that is, you do not want the operating system to delete it at the end of your job), the DD statement must name the data set and indicate how it is to be disposed of (DSNAME and DISP parameters). The DISP parameter alone will suffice if you want to use the data set in a later step but will not need it after the end of your job.

When creating a data set on a direct-access device, you must specify the amount of space required for it (SPACE parameter of DD statement).

If you want your data set stored on a particular magnetic-tape or direct-access device, you must specify the volume serial number in the DD statement (SER or REF subparameter of VOLUME parameter). If you do not specify a serial number for a magnetic-tape data set that you want to keep, the operating system will allocate one, inform the operator, and print the number on your program listing.

If your data set is to follow another data set on a magnetic-tape volume, you must use the LABEL parameter of the DD statement to indicate its sequence number on the tape.

The DCB subparameters of the DD statement that apply to consecutive data sets are listed in Figure 59 on page 159; they are described in your JCL manual. Figure 45 on page 123 shows which options of the ENVIRONMENT attribute you can specify for consecutive data sets.

ACCESSING AND UPDATING A CONSECUTIVE DATA SET

Once a consecutive data set has been created, the file that accesses it can be opened for sequential input, for sequential output, or, for data sets on direct-access devices, for update. If you open the file for output, and extend the data set by adding records at the end, DISP=MOD must be specified in the DD statement. If DISP=MOD is not specified, the data set will be overwritten. If you open a file for update, records can be updated only in their existing sequence, and if records are to be inserted, a new data set must be created. Figure 55 on page 150 shows the statements and options for accessing and updating a consecutive data set.

Subparameter	Specifies
BLKSIZE	Maximum number of bytes per block
BUFNO	Number of data management buffers
CODE	Paper tape: code in which the tape is punched
DEN	Magnetic tape: tape recording density
FUNC	Card reader or punch: function to be performed
LRECL	Maximum number of bytes per record
MODE	Card reader or punch: mode or operation (column binary or EBCDIC and Read Column Eliminate or Optical Mark Read)
OPTCD	Optional data-management services and data-set attributes
PRTSP	Printer line spacing (0, 1, 2, or 3)
RECFM	Record format and characteristics
STACK	Card reader or punch: stacker selection
TRTCH	Magnetic tape: tape recording technique for 7-track tape

Figure 59. DCB Subparameters for Consecutive Data Sets

When a consecutive data set is accessed by a SEQUENTIAL UPDATE file, a record must be retrieved with a READ statement before it can be updated by 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 permit updating a consecutive data set on magnetic tape except by adding records at the end. To replace or insert records, you must read the data set and write the updated records into a new data set.

A consecutive data set on magnetic tape can be read forward or backward. If the data set is to be read backward, the associated file must have the BACKWARDS attribute. The BACKWARDS attribute cannot be specified when a data set has V-, VB-, VS-, VBS-, D-, or DB-format records.

Parameters of DD Statement			
When Required	**Storage Device**	**What You Must State**	**Parameters**
Always		Name of data set	DSNAME=
		Disposition of data set	DISP=
If data set not cataloged	All devices	Input device	UNIT= or VOLUME=REF=
	Standard labeled magnetic tape and direct-access	Volume serial number	VOLUME=SER=
Magnetic tape: if data set not first in volume or which does not have standard labels		Sequence number	LABEL=
If data set does not have standard labels		Block size[1]	DCB=(BLKSIZE=...)

[1] Alternatively, you can specify the block size in your PL/I program by using either the ENVIRONMENT attribute or the LINESIZE option.

Figure 60. Accessing a Consecutive Data Set: Essential Parameters of DD Statement

To access a data set, you must identify it to the operating system in a DD statement. The following paragraphs, which are summarized in Figure 60, indicate the essential information you must include in the DD statement, and discuss some of the optional information you may supply. The discussions do not apply to data sets in the input stream.

Essential Information

If the data set is cataloged, you need supply only the following information in the DD statement:

- The name of the data set (DSNAME parameter). The operating system will locate the information describing the data set in the system catalog, and, if necessary, will request the operator to mount the volume containing it.
- Confirmation that the data set exists (DISP parameter). If you open the data set for output with the intention of extending it by adding records at the end, code DISP=MOD; otherwise, to open the data set for output will result in it being overwritten.

If the data set is not cataloged, you must, in addition, specify the device that will read the data set and, for magnetic-tape and direct-access devices, give the serial number of the volume that contains the data set (UNIT and VOLUME parameters).

If the data set is on paper tape or punched cards, you must specify the block size either in your PL/I program (ENVIRONMENT attribute) or in the DD statement (BLKSIZE subparameter).

If the data set follows another data set on a magnetic-tape volume, you must use the LABEL parameter of the DD statement to indicate its sequence number on the tape.

Magnetic Tape Without IBM Standard Labels

If a magnetic-tape data set has nonstandard labels or is unlabeled, you must specify the block size either in your PL/I program (ENVIRONMENT attribute) or in the DD statement (BLKSIZE subparameter). The DSNAME parameter is not essential if the data set is not cataloged.

PL/I includes no facilities for processing nonstandard labels which to the operating system appear as data sets preceding or following your data set. You can either process the labels as independent data sets or use the LABEL parameter of the DD statement to bypass them. To bypass the labels, code LABEL=(2,NL) or LABEL=(,BLP).

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 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 OF CONSECUTIVE DATA SETS

Creating and accessing consecutive data sets are illustrated in the program of Figure 61 on page 162. The program merges the contents of two data sets, in the input stream, and writes them onto a new data set, HPU8.DS3; each of the original data sets contains 15-byte fixed-length records arranged in EBCDIC collating sequence. The two input files, IN1 and IN2, have the default attribute BUFFERED, and locate mode is used to read records from the associated data sets into the respective buffers.

```
//EX8#5 JOB
//STEP1 EXEC PLIXCLG
//PLI.SYSIN  DD  *
 MERGE: PROC OPTIONS(MAIN);
        DCL  (IN1,IN2,OUT) FILE RECORD SEQUENTIAL,
             (ITEM1 BASED(A),ITEM2 BASED(B)) CHAR(15);

             ON ENDFILE(IN1) BEGIN;
              ON ENDFILE(IN2) GO TO FINISH;
 NEXT2:       WRITE FILE(OUT) FROM(ITEM2);
              PUT FILE(SYSPRINT) SKIP EDIT ('2',ITEM2) (A(2),A);
              READ FILE(IN2) SET(B);
              GO TO NEXT2;
             END;

             ON ENDFILE(IN2) BEGIN;
              ON ENDFILE(IN1) GO TO FINISH;
 NEXT1:       WRITE FILE(OUT) FROM(ITEM1);
              PUT FILE(SYSPRINT) SKIP EDIT ('1',ITEM1) (A(2),A);
              READ FILE(IN1) SET(A);
              GO TO NEXT1;
             END;

             OPEN FILE(IN1) INPUT,FILE(IN2) INPUT,FILE(OUT) OUTPUT;
             READ FILE(IN1) SET(A);
             READ FILE(IN2) SET(B);
 NEXT:       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(IN2) 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(IN1) SET(A);
              END;
             GO TO NEXT;

 FINISH:  CLOSE FILE(IN1),FILE(IN2),FILE(OUT);
          PUT FILE(SYSPRINT) PAGE;
          OPEN FILE(OUT) SEQUENTIAL INPUT;
          ON ENDFILE(OUT) GO TO FINISHPRT;
 PRINTIN:  READ FILE(OUT) INTO(ITEM1);
           PUT FILE(SYSPRINT) SKIP EDIT (ITEM1) (A);
           GO TO PRINTIN;

 FINISHPRT: CLOSE FILE(OUT);
  END MERGE;
/*
//GO.IN1 DD *
AAAAAA
CCCCCC
EEEEEE
GGGGGG
IIIIII
/*
//GO.IN2 DD *
BBBBBB
DDDDDD
FFFFFF
HHHHHH
JJJJJJ
KKKKKK
/*
//GO.OUT DD  DSN=HPU8.DS3,DISP=(NEW,CATLG),UNIT=SYSDA,
//           DCB=(RECFM=FB,BLKSIZE=150,LRECL=15),SPACE=(TRK,(1,1))
```

Figure 61. Creating and Accessing a Consecutive Data Set

You cannot use a PRINT file for record-oriented data transmission. You can still exercise some control over the layout of printed output by including a print control character as the first byte of each of your output records; you can also use similar control characters to select the stacker to which cards punched by your program are fed.

The operating system recognizes two types of control characters for printer and card punch commands — American National Standard control characters and machine code control characters. You must indicate which control character you are using, either in your PL/I program (ENVIRONMENT attribute CTL360 or CTLASA option), or in the DD statement (RECFM subparameter). If you specify one of these characters, but transmit your data to a device other than a printer or a card punch, the operating system transmits the control characters as part of your records. If you use an invalid control character, "Space 1 line" or "Select stacker 1" is the default.

The American National Standard control characters, which are listed in Figure 62, cause the specified action to occur before the associated record is printed or punched.

The machine code control characters differ according to the type of device. The IBM machine code control characters for printers are listed in Figure 63 on page 164.

Code	Action
b	Space 1 line before printing (blank code)
0	Space 2 lines before printing
-	Space 3 lines before printing
+	Suppress space before printing
1	Skip to channel 1
2	Skip to channel 2
3	Skip to channel 3
4	Skip to channel 4
5	Skip to channel 5
6	Skip to channel 6
7	Skip to channel 7
8	Skip to channel 8
9	Skip to channel 9
A	Skip to channel 10
B	Skip to channel 11
C	Skip to channel 12
V	Select stacker 1
W	Select stacker 2

Figure 62. American National Standard Print and Card Punch Control Characters (CTLASA)

Figure 64 on page 164 gives control codes for the IBM 2540 Card Read Punch. Control codes for the IBM 3525 Card Printer are given in Figure 65 on page 164 and Figure 66 on page 165.

Print and Then Act	Action	Act Immediately (no printing)
Code Byte		Code Byte
00000001	Print only (no space)	-
00001001	Space 1 line	00001011
00010001	Space 2 lines	00010011
00011001	Space 3 lines	00011011
10001001	Skip to channel 1	10001011
10010001	Skip to channel 2	10010011
10011001	Skip to channel 3	10011011
10100001	Skip to channel 4	10100011
10101001	Skip to channel 5	10101011
10110001	Skip to channel 6	10110011
10111001	Skip to channel 7	10111011
11000001	Skip to channel 8	11000011
11001001	Skip to channel 9	11001011
11010001	Skip to channel 10	11010011
11011001	Skip to channel 11	11011011
11100001	Skip to channel 12	11100011

Figure 63. IBM Machine Code Print Control Characters (CTL360)

Code Byte	Action
00000001	Select stacker 1
01000001	Select stacker 2
10000001	Select stacker 3

Figure 64. 2540 Card Read Punch Control Characters (CTL360)

Code byte	Action
00001101	Print on line 1
00010101	Print on line 2
00011101	Print on line 3
00100101	Print on line 4
00101101	Print on line 5
00110101	Print on line 6
00111101	Print on line 7
01000101	Print on line 8
01001101	Print on line 9
01010101	Print on line 10
01011101	Print on line 11
01100101	Print on line 12
01101101	Print on line 13
01110101	Print on line 14
01111101	Print on line 15
10000101	Print on line 16
10001101	Print on line 17
10010101	Print on line 18
10011101	Print on line 19
10100101	Print on line 20
10101101	Print on line 21
10110101	Print on line 22
10111101	Print on line 23
11000101	Print on line 24
11001101	Print on line 25

Figure 65. 3525 Card Printer Control Characters (CTL360)

Code	Action
b	Space 1 line and print
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

Figure 66. 3525 Card Printer Control-Characters (CTLASA)

There are two types of machine-code control characters for the printer — one causing the action to occur after the record has been transmitted, and the other producing immediate action but transmitting no data (include the second type only in a blank record).

The essential requirements for producing printed output or punched cards are exactly the same as those for creating any other consecutive data set (described above). For a printer, if you do not use one of the control characters, all data will be printed sequentially, with no spaces between records; each block will be interpreted as the start of a new line. When you specify a block size for a printer or card punch, and are using one of the control characters, allow for the control character in your block size; for example, if you want to print lines of 100 characters, specify a block size of 101.

Example

The program in Figure 67 on page 166 uses record-oriented data transmission to read and print the contents of the data set SINES, created by the PRINT file in Figure 52 on page 145. Since the data set SINES is cataloged, only two parameters are required in the DD statement that defines it. The output file PRINTER is declared with the ENVIRONMENT option CTLASA, specifying that the first byte of each record will be interpreted as an American National Standard print control character. The other information given in the ENVIRONMENT attribute could alternatively have been given in the DD statement, as follows:

```
DCB=(RECFM=VA,BLKSIZE=102)
```

```
//EX8#11  JOB
//STEP1 EXEC PLIXCLG
//PLI.SYSIN  DD  *
 PRT:    PROC OPTIONS(MAIN);
         DCL TABLE FILE RECORD INPUT SEQUENTIAL,
          PRINTER FILE RECORD OUTPUT SEQL ENV(V BLKSIZE(102) CTLASA),
             LINE CHAR(94) VAR;
         ON ENDFILE(TABLE) GO TO FINISH;
         OPEN FILE(TABLE),FILE(PRINTER);
 NEXT:   READ FILE(TABLE) INTO(LINE);
         WRITE FILE(PRINTER) FROM(LINE);
         GO TO NEXT;
 FINISH: CLOSE FILE(TABLE),FILE(PRINTER);
 END PRT;
/*
//GO.TABLE DD DSNAME=HPU8.SINES,DISP=(OLD,DELETE),UNIT=SYSDA,
//      SPACE=(TRK,(1,1)),VOL=SER=nnnnnn
//GO.PRINTER DD SYSOUT=A
```

Figure 67. Printing with Record-Oriented Data Transmission

DEVICE-ASSOCIATED FILES (IBM 3525 CARD PUNCH)

The IBM 3525 is an 80-column card punch, available to IBM System/370 users, that can also read cards and print on them. The CTLASA and CTL360 control characters for the device are given earlier in "Punching Cards and Printing" on page 163.

You can use the multiple capabilities of the device by associating two or three files together with the device so that more than one of the operations read, punch, and print can be performed on the same card during one pass through the device. Details of the use of the device, together with the IBM 3505 card reader, are given in "IBM 3505 and 3525 Card Reader and Punch" on page 108. However, you must consider the following restrictions at the time you write the program.

- Device-associated files must have the RECORD attribute and must be either all BUFFERED or all UNBUFFERED.
- The records must be F-format. The maximum record size is 80 for read and punch files and 64 for print files, plus 1 byte for punch/print control characters.
- ENVIRONMENT(TOTAL) cannot be used.
- When a read or punch associated file is opened, the value of the BUFFERS option (for BUFFERED files) or of NCP (for UNBUFFERED files) will be set to one.
- Device-associated files may be opened in any order, but all of the files must be open before any transmission takes place to or from any one of them.
- Depending on the files associated, the appropriate input/output operations on each card must strictly follow the order read-punch-print. If the sequence rules are not followed, the ERROR condition is raised. Only the print operation can be omitted or repeated.
- A print-associated file that uses control characters for line positioning must not attempt to feed a card. Such an attempt would occur if an instruction to print beyond the maximum line number (2 or 25) for the card were used, or if a control character that implied a new record were used. For example, the control character '1' specifies printing on the first line of the next card.

- Device-associated files can normally be closed in any order, but no transmission can take place after any one of the files has been closed. As a result, care is needed if the LOCATE statement is used for BUFERED OUTPUT files. The output from a LOCATE statement does not actually take place until the next LOCATE, WRITE, or CLOSE statement for the file. If the LOCATE statement is used on both print- and punch-associated files, a multiple CLOSE statement must be used, specifying the punch file before the print file. For example:

```
LOCATE A FILE(PUNCHOUT);
LOCATE B FILE(PRINTOUT);
CLOSE
FILE(PUNCHOUT),FILE(PRINTOUT);
```

- The American National Standard print control character '+' (or SKIP(0)) is not allowed with the IBM 3525.
- Files associated with column binary or Optical Mark Read data sets must be RECORD files.

INDEXED DATA SETS

This section describes indexed data set organization and the data transmission statements and ENVIRONMENT options that define indexed data sets. It then describes how to create, access, and reorganize indexed data sets.

INDEXED ORGANIZATION

A data set with indexed organization must be on a direct-access device. Its records, which can be either F-format or V-format records, blocked or unblocked, are arranged in logical sequence according to keys that are associated with each record. A key is a character string that can identify each record uniquely. Logical records are arranged in the data set in ascending key sequence according to the EBCDIC collating sequence. Indexes associated with the data set are used by the operating system data-management routines to locate a record when the key is supplied.

Unlike consecutive organization, indexed organization does not require every record to be accessed in sequential fashion. An indexed data set must be created sequentially; but, once it has been created, the associated file may be opened for SEQUENTIAL or DIRECT access, as well as INPUT or UPDATE. When the file has the DIRECT attribute, records may be retrieved, added, deleted, and replaced at random.

Sequential processing of an indexed data set is slower than that of a corresponding consecutive data set, because the records it contains are not necessarily retrieved in physical sequence; furthermore, random access is less efficient for an indexed data set than for a regional data set, because the indexes must be searched to locate a record. An indexed data set requires more external storage space than a consecutive data set, and all volumes of a multivolume data set must be mounted, even for sequential processing.

Figure 68 on page 167 list the data-transmission statements and options that can be used to create and access an indexed data set.

File declaration[1]	Valid statements with options that must appear	Other options that can also be used
SEQUENTIAL OUTPUT	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	
	LOCATE based-variable FILE(file-reference) KEYFROM(expression);	SET(pointer-reference)
SEQUENTIAL INPUT	READ FILE(file-reference) INTO(reference);	KEY(expression) or KEYTO(reference)
	READ FILE(file-reference) SET(pointer-reference);	KEY(expression) or KEYTO(reference)
	READ FILE(file-reference) IGNORE(expression);	
SEQUENTIAL UPDATE	READ FILE(file-reference) INTO(reference);	KEY(expression) or KEYTO(reference)
	READ FILE(file-reference) SET(pointer-reference);	KEY(expression) or KEYTO(reference)
	READ FILE(file-reference) IGNORE(expression);	
	REWRITE FILE(file-reference);	FROM(reference)
	DELETE FILE(file-reference);[2]	KEY(expression)
DIRECT INPUT	READ FILE(file-reference) INTO(reference) KEY(expression);	EVENT(event-reference)
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(rile-reference) FROM(reference) KEYFROM(expression);	EVENT(event-reference)
	DELETE FILE(file-reference) KEY(expression);[2]	EVENT(event-reference)

Figure 68 (Part 1 of 2). Statements and Options Permitted for Creating and Accessing Indexed Data Sets

File declaration[1]	Valid statements with options that must appear	Other options that can also be used
DIRECT UPDATE EXCLUSIVE	READ FILE(file-reference) INTO(reference) KEY(expression);	EVENT(event-reference) and/or NOLOCK
	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);[2]	EVENT(event-reference)
	UNLOCK FILE(file-reference) KEY(expression)	

Figure 68 (Part 2 of 2). Statements and Options Permitted for Creating and Accessing Indexed Data Sets

Notes to Figure 68:

1 The complete file declaration would include the attributes FILE, RECORD, and ENVIRONMENT. If any of the options KEY, KEYFROM, or KEYTO are used, the file declaration must also include the attribute KEYED. The attribute BUFFERED is the default, and UNBUFFERED is ignored for INDEXED SEQUENTIAL and SEQUENTIAL files.

2 Use of the DELETE statement is invalid if OPTCD=L (DCB subparameter) was not specified when the data set was created or if the RKP subparameter is 0 for FB records, or 4 for V and VB records.

Keys

There are two kinds of keys—recorded keys and source keys. A recorded key is a character string that actually appears with each record in the data set to identify that record; its length cannot exceed 255 characters and all keys in a data set must have the same length. The recorded keys in an indexed data set may be separate from, or embedded within, the logical records. A source key is the character value of the expression that appears in the KEY or KEYFROM option of a data transmission statement to identify the record to which the statement refers; for direct access of an indexed data set, each transmission statement must include a source key.

Note: All VSAM key-sequenced data sets have embedded keys, even if they have been converted from ISAM data sets with nonembedded keys.

Embedded Keys

The use of embedded keys avoids the need for the KEYTO option during sequential input, but the KEYFROM option is still required for output. (However, the data specified by the KEYFROM option may be the embedded key portion of the record variable itself.) In a data set with unblocked records, a separate recorded key precedes each record, even when there is already an embedded key. If the records are blocked, the key of only the last record in each block is recorded separately in front of the block.

During the execution of a WRITE statement that adds a record to a data set with embedded keys, the value of the expression in the KEYFROM option is assigned to the embedded key position in the record variable. Note that a record variable can be declared as a structure with an embedded key declared as a structure member, but that such an embedded key must not be declared as a VARYING string.

For a LOCATE statement, the KEYFROM string is assigned to the embedded key when the next operation on the file is encountered.

Indexes

To provide faster access to the records in the data set, the operating system creates and maintains a system of indexes to the records in the data set. The lowest level of index is the track index. There is a track index for each cylinder in the data set; it occupies the first track (or tracks) of the cylinder, and lists the key of the last record on each track in the cylinder. A search can then be directed to the first track that has a key that is higher than or equal to the key of the required record.

If the data set occupies more than one cylinder, the operating system develops a higher-level index called a cylinder index. Each entry in the cylinder index identifies the key of the last record in the cylinder. To increase the speed of searching the cylinder index, you can request in a DD statement that the operating system develop a master index for a specified number of cylinders; you can have up to three levels of master index; Figure 69 on page 171 illustrates the index structure. The part of the data set that contains the cylinder and master indexes is termed the index area.

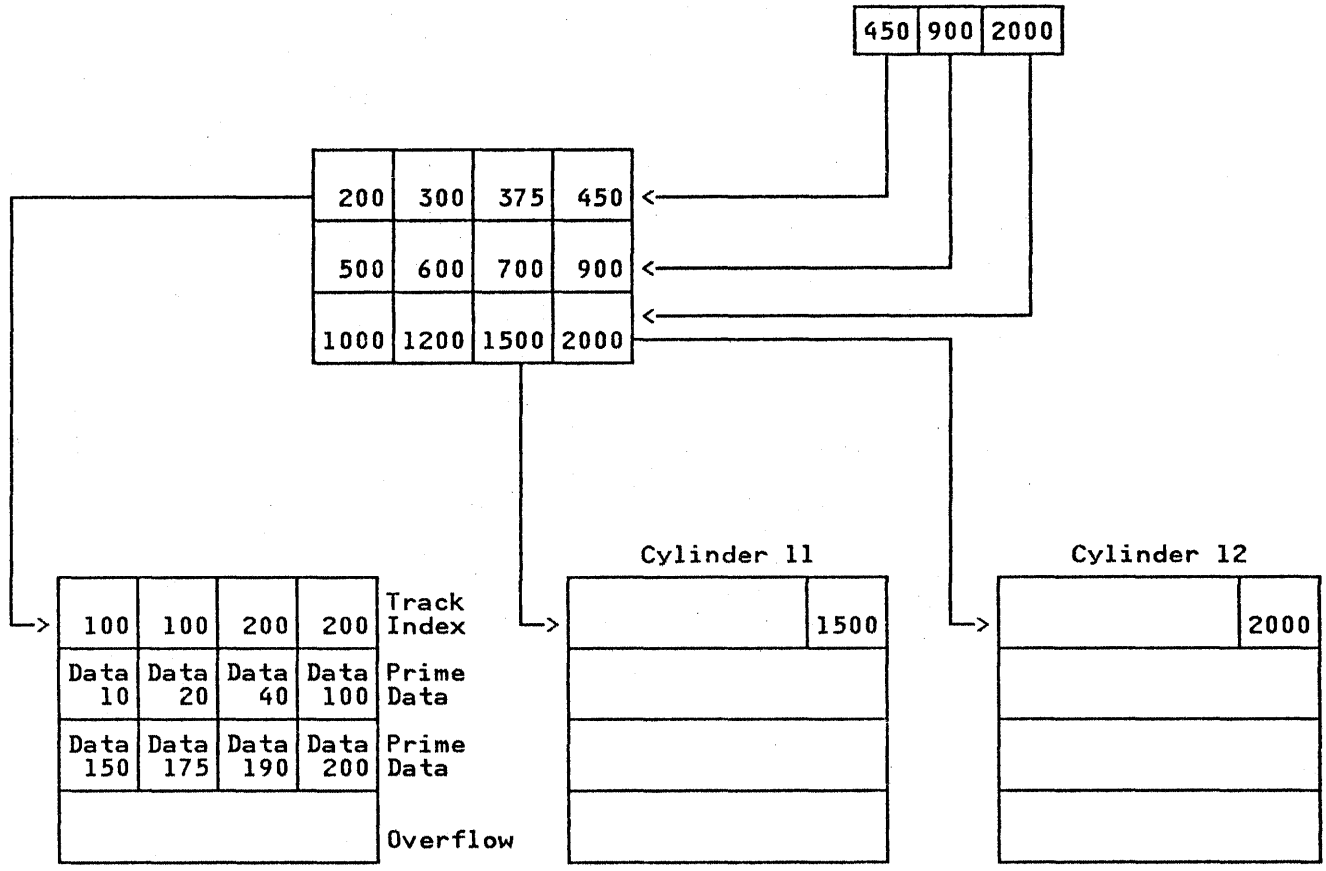


Figure 69. Index Structure of An Indexed Data Set

When an indexed data set is created, all the records are written in what is called the prime data area. If more records are added later, the operating system does not rearrange the entire data set; it inserts each new record in the appropriate position and moves up the other records on the same track. Any records forced off the track by the insertion of a new record are placed in an overflow area. The overflow area can consist either of a number of tracks set aside in each cylinder for the overflow records from that cylinder (cylinder overflow area), or a separate area for all overflow records (independent overflow area).

Records in the overflow area are chained together to the track index so as to maintain the logical sequence of the data set; this is illustrated in Figure 70 on page 174. Each entry in the track index consists of two parts:

- The normal entry, which points to the last record on the track
- The overflow entry, which contains the key of the first record transferred to the overflow area and also points to the last record transferred from the track to the overflow area

If there are no overflow records from the track, both index entries point to the last record on the track. An additional field is added to each record that is placed in the overflow area. It points to the previous record transferred from the same track; the first record from each track is linked to the corresponding overflow entry in the track index.

Dummy Records

Records within an indexed data set are either actual records containing valid data or dummy records. A dummy record, identified by the constant (8)'1'B in its first byte, can be one that you insert or it can be created by the operating system. You insert dummy records by setting the first byte to (8)'1'B and writing the records in the usual way. The operating system creates dummy records by placing (8)'1'B in a record that is named in a DELETE statement.

When creating an indexed data set, you may wish to insert dummy records to reserve space in the prime data area. Dummy records can later be replaced by valid data records having the same key.

The operating system removes dummy records when the data set is reorganized, as described later in this section, and removes those forced off the track during an update.

If the DCB subparameter OPTCD=L is included in the DD statement that defines the data set when it is created, dummy records will not be retrieved by READ statements and the operating system will write the dummy identifier in records being deleted.

DEFINING AN INDEXED DATA SET

A sequential indexed data set is defined by a file declaration with the following attributes:

```
DCL filename FILE RECORD
             INPUT | OUTPUT | UPDATE
             SEQUENTIAL
             BUFFERED
            [KEYED]
             ENVIRONMENT(options);
```

A direct indexed data set is defined by a file declaration with the following attributes:

```
DCL filename FILE RECORD
             INPUT | OUTPUT | UPDATE
             DIRECT
             UNBUFFERED
             KEYED
            [EXCLUSIVE]
             ENVIRONMENT(options);
```

Default file attributes are shown in Figure 45 on page 123. The file attributes are described in the OS and DOS PL/I Language Reference Manual. Options of the ENVIRONMENT attribute are discussed below.

ENVIRONMENT OPTIONS FOR INDEXED DATA SETS

The ENVIRONMENT options applicable to indexed data sets are:

```
F|FB|V|VB
RECSIZE(record-length)
BLKSIZE(block-size)
SCALARVARYING
COBOL
BUFFERS(n)
KEYLENGTH(n)
NCP(n)
GENKEY

INDEXED
KEYLOC(n)
INDEXAREA[(index-area-size)]
NOWRITE
ADDBUFF
```

The options above the blank line are described in "Data Set Organization Options" on page 122, and those below the blank line are described below.

INDEXED Option

The INDEXED option defines a file with indexed organization (which is described above). It is usually used with a data set created and accessed by the Indexed Sequential Access Method, but may also be used in some cases with VSAM data sets (as described in Chapter 9).

Syntax

```
INDEXED
```

KEYLOC Option — Key Location

The KEYLOC option can be used with indexed data sets, when the data set is created, to specify the starting position of an embedded key in a record.

Syntax

```
KEYLOC(n)
```

The position, n, must be within the limits:

$$1 \leq n \leq \text{recordsize} - \text{keylength} + 1$$

That is, the key cannot be larger than the record, and must be contained completely within the record.

If the keys are embedded within the records, either the KEYLOC(n) option should be specified, or the DCB subparameter RKP must be included in the DD statement for the associated data set.

If KEYLOC is not specified, the value specified with RKP is used. If this subparameter is not specified, then RKP=0 is the default.

The KEYLOC option specifies the absolute position of an embedded key from the start of the data in a record, while the RKP subparameter specifies the position of an embedded key relative to the start of the record.

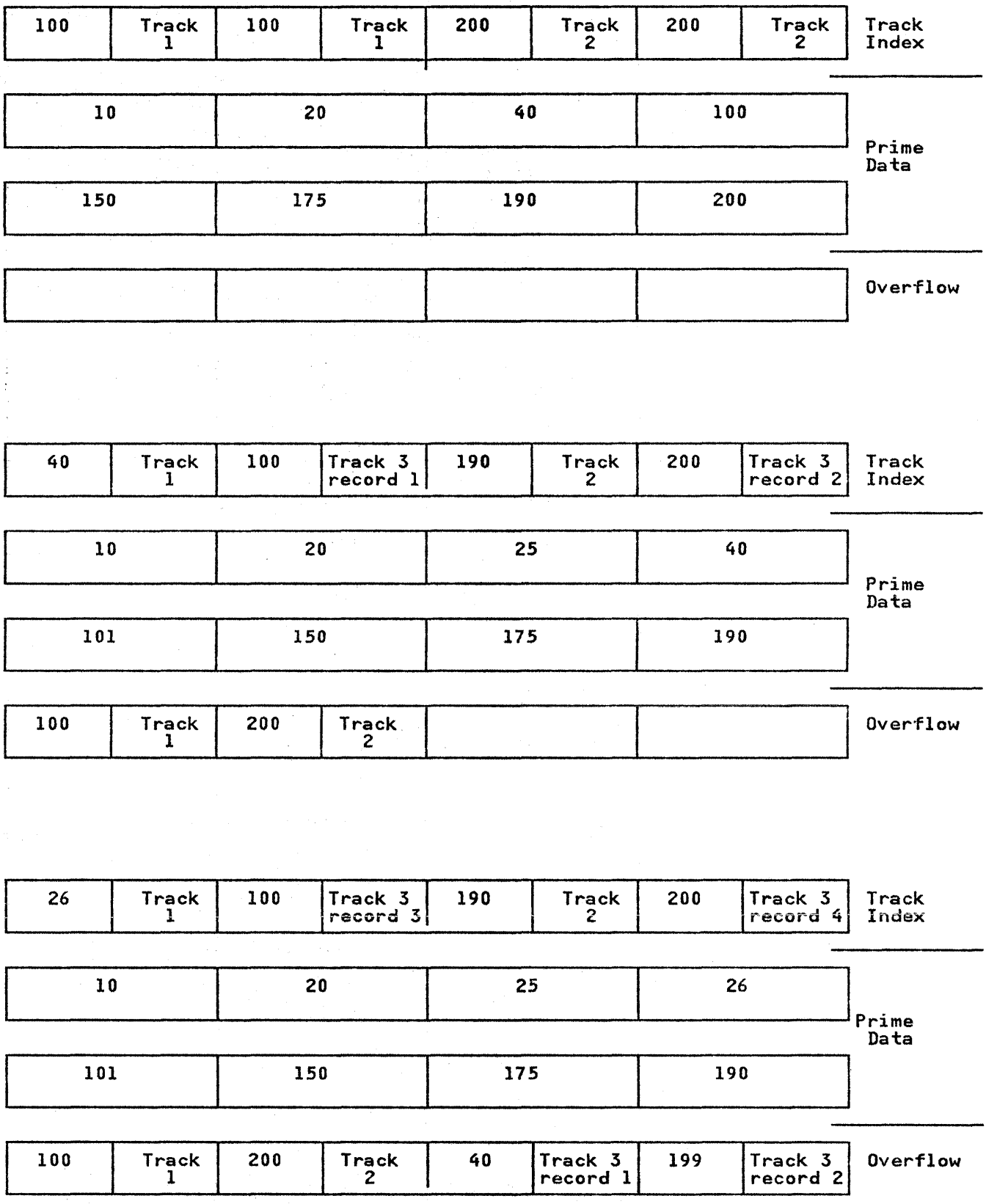


Figure 70. Adding Records to an Indexed Data Set

Thus the equivalent KEYLOC and RKP values for a particular byte are affected by the following:

- The KEYLOC byte count starts at 1; the RKP count starts at 0
- The record format

For example, if the embedded key begins at the tenth byte of a record variable, then the specifications are:

```
Fixed length:      KEYLOC(10)
                   RKP=9

Variable-length:   KEYLOC(10)
                   RKP=13
```

If KEYLOC is specified with a value equal to or greater than 1, embedded keys exist in the record variable and on the data set. If KEYLOC is equal to zero, or is not specified, the RKP value is used; when RKP is specified, the key is part of the variable only when RKP≥1. As a result, embedded keys may not always be present in the record variable or the data set. If KEYLOC(1) is specified, it must be specified for every file that accesses the data set. This is necessary because KEYLOC(1) cannot be converted to an unambiguous RKP value. (Its equivalent is RKP=0 for fixed format, which in turn implies nonembedded keys.) The effect of the use of both options is shown in Figure 71.

If SCALARVARYING is specified, the embedded key must not immediately precede or follow the first byte; hence, the value specified for KEYLOC must be greater than 2.

If the KEYLOC option is included in a VSAM file declaration for checking purposes, and the key location specified in the option conflicts with the value defined for the data set, the UNDEFINEDFILE condition is raised.

KEYLOC(n)	RKP	Record Variable	Data Set Unblocked Records	Data Set Blocked Records
n>1	RKP equivalent = n-1+C[1]	Key	Key	Key
n=1	No equivalent	Key	Key[2]	Key
n=0 or not specified	RKP=C[1]	No Key	No Key	Key[3]
	RKP>C[1]	Key	Key	Key

Figure 71. Effect of KEYLOC and RKP Values on Establishing Embedded Keys in-Record Variables or Data Sets

Notes to Figure 71:

1 C = number of control bytes, if any:

```
C=0 for fixed-length records
C=4 for variable-length records
```

2 In this instance the key is not recognized by data management.

3 Each logical record in the block has a key.

INDEXAREA Option

The INDEXAREA option improves the input/output speed of a DIRECT INPUT or DIRECT UPDATE file with indexed data set organization, by having the highest level of index placed in main storage.

Syntax

```
INDEXAREA[(index-area-size)]
```

The "index-area-size" enables you to limit the amount of main storage allowed for an index area. The size, when specified, must be an integer or a variable with attributes FIXED BINARY(31,0) STATIC whose value lies within the range 0 through 64,000. If the "index-area-size" is not specified, the highest level index is moved unconditionally into main storage. If an "index-area-size" is specified, the highest level index is held in main storage, provided that its size does not exceed that specified. If the specified size is less than 0 or greater than 64,000, unpredictable results will occur.

NOWRITE Option

The NOWRITE option is used for DIRECT UPDATE files. It specifies that no records are to be added to the data set and that data management modules concerned solely with adding records are not required; it thus allows the size of the object program to be reduced.

Syntax

```
NOWRITE
```

ADDBUFF Option

The ADDBUFF option can be specified for a DIRECT INPUT or DIRECT UPDATE file with indexed data set organization and F-format records to indicate that an area of internal storage is to be used as a workspace in which records on the data set can be rearranged when new records are added. The size of the workspace is equivalent to one track of the direct-access device used. The option need not be specified for DIRECT INDEXED files with V-format records, as the workspace is automatically allocated for such files.

Syntax

```
ADDBUFF
```

CREATING AN INDEXED DATA SET

When you create an indexed data set, the associated file must be opened for SEQUENTIAL OUTPUT, and the records must be presented in the order of ascending key values. (If there is an error in the key sequence, the KEY condition is raised.) A DIRECT file cannot be used for the creation of an indexed data set.

Figure 68 on page 167 shows the statements and options for creating an indexed data set.

An indexed data set consisting of fixed-length records can be extended by adding records sequentially at the end, until the original space allocated for the prime data is filled. The corresponding file must be opened for SEQUENTIAL OUTPUT and you must include DISP=MOD in the DD statement.

You can use a single DD statement to define the whole of the data set (index area, prime area, and overflow area), or you can use two or three statements to define the areas independently. If you use two DD statements, you can define either the index area and the prime area together, or the prime area and the overflow area together.

If you want the whole of the data set to be on a single volume, there is no advantage to be gained by using more than one DD statement except to define an independent overflow area (see "Overflow Area" on page 182). But, if you use separate DD statements to define the index and/or overflow area on volumes separate from that which contains the prime area, you will increase the speed of direct-access to the records in the data set by reducing the number of access mechanism movements required.

When you use two or three DD statements to define an indexed data set, the statements must appear in the order: index area; prime area; overflow area. The first DD statement must have a name (ddname), but the name fields of a second or third DD statement must be blank. The DD statements for the prime and overflow areas must specify the same type of unit (UNIT parameter). You must include all the DCB information for the data set in the first DD statement; DCB=DSORG=IS will suffice in the other statements.

Essential Information

To create an indexed data set, you must give the operating system certain information either in your PL/I program or in the DD statement that defines the data set. The following paragraphs indicate the essential information, and discuss some of the optional information you may supply.

You must supply the following information when creating an indexed data set:

- Direct-access device that will write your data set (UNIT or VOLUME parameter of DD statement). Do not request DEFER.
- Block size: You can specify the block size either in your PL/I program (ENVIRONMENT attribute or LINESIZE option) or in the DD statement (BLKSIZE subparameter). If you do not specify a record length, unblocked records are the default and the record length is determined from the block size.
- Space requirements: Include space for future needs when you specify the size of the prime, index, and overflow areas. Once you have created an indexed data set, you cannot change its specification.

If you want to keep a direct-access data set (that is, you do not want the operating system to delete it at the end of your job), the DD statement must name the data set and indicate how it is to be disposed of (DSNAME and DISP parameters). The DISP parameter alone will suffice if you want to use the data set in a later step but will not need it after the end of your job.

If you want your data set stored on a particular direct-access device, you must specify the volume serial number in the DD statement (SER or REF subparameter of VOLUME parameter). If you do not specify a serial number for a data set that you want to keep, the operating system will allocate one, inform the operator, and print the number on your program listing. All the essential parameters required in a DD statement for the creation of an indexed data set are summarized in Figure 72 on page 178. Figure 73 on page 179 lists the DCB subparameters needed. See your JCL manual for a description of the DCB subparameters.

Parameters of DD Statement

When Required	What You Must State	Parameters
Always	Output device	UNIT= or VOLUME=REF=
	Storage space required	SPACE=
	Data control block information: see Figure 73 on page 179.	DCB=
More than one DD statement	Name of data set and area (index, prime, overflow)	DSNAME=
Data set to be used in another job step but not required after end of job	Disposition	DISP=
Data set to be kept after end of job	Disposition	DISP=
	Name of data set	DSNAME=
Data set to be on particular volume	Volume serial number	VOLUME=SER= or VOLUME=REF=

Figure 72. Creating an Indexed Data Set: Essential Parameters of DD Statement

You must request space for the prime data area in the SPACE parameter. You cannot specify a secondary quantity for an indexed data set. Your request must be in units of cylinders unless you place the data set in a specific position on the volume (by specifying a track number in the SPACE parameter). In the latter case, the number of tracks you specify must be equivalent to an integral number of cylinders, and the first track must be the first track of a cylinder other than the first cylinder in the volume. You can also use the SPACE parameter to specify the amount of space to be used for the cylinder and master indexes (unless you use a separate DD statement for this purpose). If you do not specify the space for the indexes, the operating system will use part of the independent overflow area; if there is no independent overflow area, it will use part of the prime data area.

You must always specify the data set organization (DSORG=IS subparameter of the DCB parameter), and in the first (or only) DD statement you must also specify the length of the key (KEYLEN subparameter of the DCB parameter) unless it is specified in the ENVIRONMENT attribute.

If you want the operating system to recognize dummy records, you must code OPTCD=L in the DCB subparameter of the DD statement. This will cause the operating system to write the dummy identifier in deleted records and to ignore dummy records during sequential read processing. Do not specify OPTCD=L when using blocked or variable-length records with nonembedded keys; if you do, the dummy record identifier (8)'1'B will overwrite the key of deleted records.

When Required	DCB Subparameters To Specify	Subparameters
These are always required[2]	Record format[1]	RECFM=F, FB, V, or VB
	Block size[1]	BLKSIZE=
	Data set organization	DSORG=IS
	Key length[1]	KEYLEN=
Include at least one of these if overflow is required	Cylinder overflow area and number of tracks per cylinder for overflow records	OPTCD=Y and CYLOFL=
	Independent overflow area	OPTCD=I
These are optional	Record length[1]	LRECL=
	Embedded key (relative key position)[1]	RKP= [2]
	Master index	OPTCD=M
	Automatic processing of dummy records	OPTCD=L
	Number of data management buffers[1]	BUFNO=
	Number of tracks in cylinder index for each master index entry	NTM=

[1] Alternatively, can be specified in ENVIRONMENT attribute.

[2] RKP is required if the data set has embedded keys, unless the KEYLOC option of ENVIRONMENT is specified instead.

Note: Full DCB information must appear in the first, or only, DD statement. Subsequent statements require only DSORG=IS.

Figure 73. DCB Subparameters for an Indexed Data Set

You cannot place an indexed data set on a system output (SYSOUT) device.

Name of the Data Set

If you use only one DD statement to define your data set, you need not name the data set unless you intend to access it in another job. But, if you include two or three DD statements, you must specify a data set name, even for a temporary data set.

The DSNAME parameter in a DD statement that defines an indexed data set not only gives the data set a name, but it also identifies the area of the data set to which the DD statement refers:

```
DSNAME=name(INDEX)
DSNAME=name(PRIME)
DSNAME=name(OVFLOW)
```

If you use one DD statement to define the prime and index or one DD statement to define the prime and overflow area, code DSNAME=name(PRIME). If you use one DD statement for the entire file (prime, index, and overflow), code DSNAME=name(PRIME) or simply DSNAME=name.

Record Format and Keys

An indexed data set can contain either fixed- or variable-length records, blocked or unblocked. You must always specify the record format, either in your PL/I program (ENVIRONMENT attribute) or in the DD statement (RECFM subparameter).

The key associated with each record can be contiguous with or embedded within the data in the record.

If the records are unblocked, the key of each record is recorded in the data set in front of the record even if it is also embedded within the record, as shown in (a) and (b) of Figure 74 on page 181. If blocked records do not have embedded keys, the key of each record is recorded within the block in front of the record, and the key of the last record in the block is also recorded just ahead of the block, as shown in (c) of Figure 74 on page 181. When blocked records have embedded keys, the individual keys are not recorded separately in front of each record in the block; the key of the last record in the block is recorded in front of the block, as shown in (d) of Figure 74 on page 181.

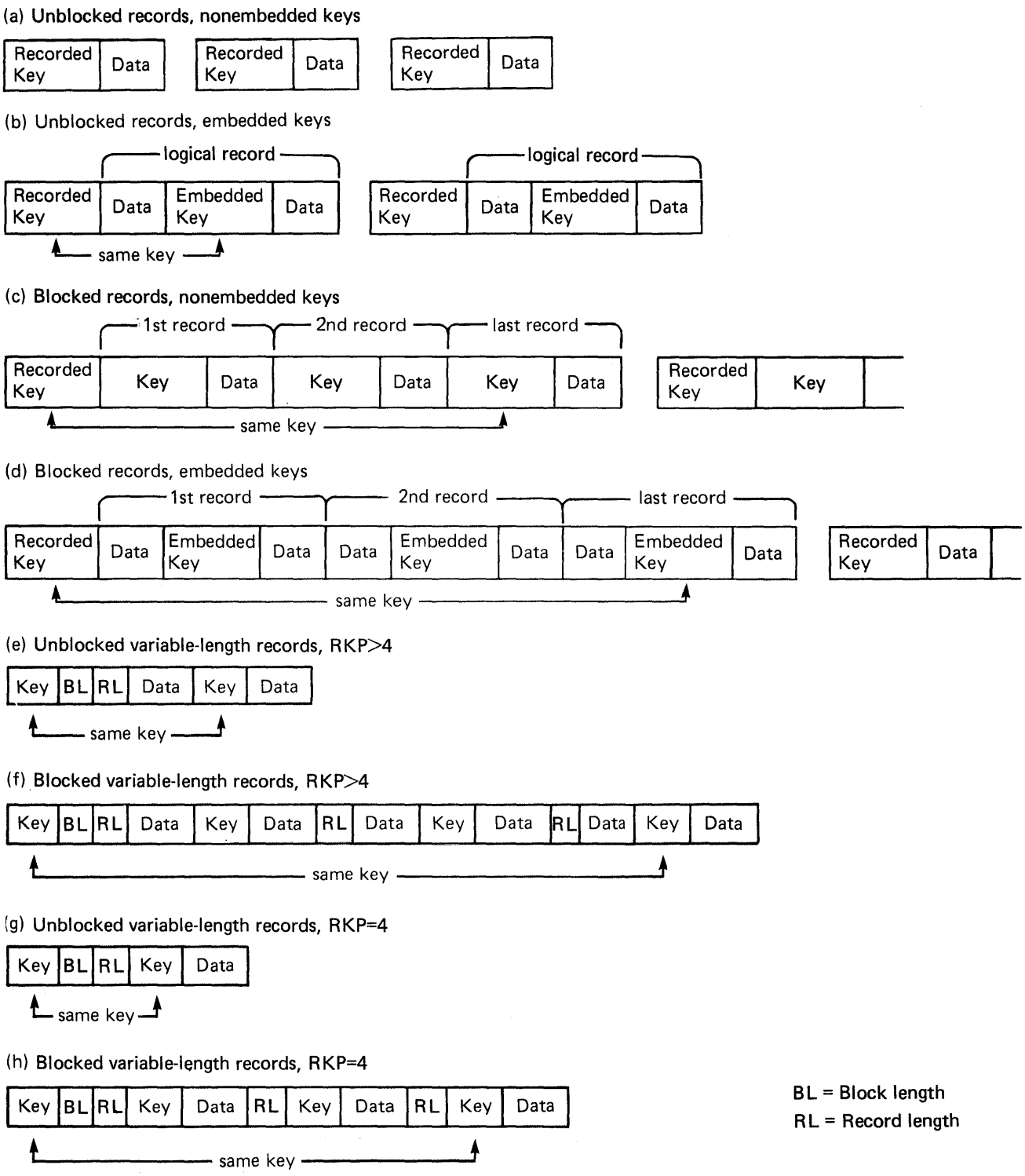


Figure 74. Record Formats in an Indexed Data Set

If you use blocked records with nonembedded keys, the record size that you specify must include the length of the key, and the block size must be a multiple of this combined length. Otherwise, record length and block size refer only to the data in the record. Record format information is shown in Figure 75.

RECORDS	RKP	LRECL	BLKSIZE
Blocked	Not zero	R	R * B
	Zero or omitted	R + K	B*(R+K)
Unblocked	Not zero	R	R
	Zero or omitted	R	R

R = Size of data in record

K = Length of keys (as specified in KEYLEN subparameter)

B = Blocking factor

Example:
For blocked records, nonembedded keys, 100 bytes of data per record, 10 records per block, key length = 20:
LRECL=120,BLKSIZE=1200,RECFM=FB

Figure 75. Record Format Information for an Indexed Data Set

If you use records with embedded keys, you must include the DCB subparameter RKP to indicate the position of the key within the record. For fixed-length records the value specified in the RKP subparameter is 1 less than the byte number of the first character of the key; that is, if RKP=1, the key starts in the second byte of the record. The default value if you omit this subparameter is RKP=0, which specifies that the key is not embedded in the record but is separate from it.

For variable-length records, the value specified in the RKP subparameter must be the relative position of the key within the record plus 4. The extra 4 bytes take into account the 4-byte control field used with variable-length records. For this reason, you must never specify RKP less than 4. When deleting records, you must always specify RKP equal to or greater than 5, since the first byte of the data is used to indicate deletion.

For unblocked records, the key, even if embedded, is always recorded in a position preceding the actual data. Consequently, the RKP subparameter need not be specified for unblocked records.

Overflow Area

If you intend to add records to the data set on a future occasion, you must request either a cylinder overflow area or an independent overflow area, or both.

For a cylinder overflow area, include the DCB subparameter OPTCD=Y and use the subparameter CYLOFL to specify the number of tracks in each cylinder to be reserved for overflow records. A cylinder overflow area has the advantage of a short search time for overflow records, but the amount of space available for overflow records is limited, and much of the space may be unused

if the overflow records are not evenly distributed throughout the data set.

For an independent overflow area, use the DCB subparameter OPTCD=I to indicate that overflow records are to be placed in an area reserved for overflow records from all cylinders, and include a separate DD statement to define the overflow area. The use of an independent area has the advantage of reducing the amount of unused space for overflow records, but entails an increased search time for overflow records.

It is good practice to request cylinder overflow areas large enough to contain a reasonable number of additional records and an independent overflow area to be used as the cylinder overflow areas are filled.

If the prime data area is not filled during creation, you cannot use the unused portion for overflow records, nor for any records subsequently added during direct-access (although you can fill the unfilled portion of the last track used). You can reserve space for later use within the prime data area by writing dummy records during creation (see "Dummy Records" on page 172).

Master Index

If you want the operating system to create a master index for you, include the DCB subparameter OPTCD=M, and indicate in the NTM subparameter the number of tracks in the cylinder index you wish to be referred to by each entry in the master index. The operating system will create up to three levels of master index, the first two levels addressing tracks in the next lower level of the master index.

ACCESSING AN INDEXED DATA SET

Once an indexed data set has been created, the file that accesses it can be opened for SEQUENTIAL INPUT or UPDATE, or for DIRECT INPUT or UPDATE. In the case of F-format records, it can also be opened for OUTPUT to add records at the end of the data set. The keys for these records must have higher values than the existing keys for that data set and must be in ascending order. Figure 68 on page 167 shows the statements and options for accessing an indexed data set.

Sequential input allows you to read the records in ascending key sequence, and in sequential update you can read and rewrite each record in turn. Using direct input, you can read records using the READ statement, and in direct update you can read or delete existing records or add new ones. Sequential and direct-access are discussed in further detail below.

Sequential Access

A sequential file that is used to access an indexed data set may be opened with either the INPUT or the UPDATE attribute. The data transmission statements need not include source keys, nor need the file have the KEYED attribute. Sequential access is in order of ascending recorded-key values; records are retrieved in this order, and not necessarily in the order in which they were added to the data set. Dummy records are not retrieved if the DD statement that defined the data set included the subparameter OPTCD=L.

Except that the EVENT option cannot be used, rules governing the relationship between the READ and REWRITE statements for a SEQUENTIAL UPDATE file that accesses an indexed data set are identical to those for a consecutive data set (described above).

Embedded keys in a record to be updated must not be altered. The modified record must always overwrite the update record in the data set.

Additionally, records can be effectively deleted from the data set; a DELETE statement marks a record as a dummy by putting (8)'1'B in the first byte. The DELETE statement should not be used to process a data set with F-format blocked records and either KEYLOC=1 or RKP=0, or V- or VB-format records and either KEYLOC=1 or RKP=4. (The code (8)'1'B would overwrite the first byte of the recorded key.) Note that the EVENT option is not supported for SEQUENTIAL access of indexed data sets.

INDEXED KEYED files opened for SEQUENTIAL INPUT and SEQUENTIAL UPDATE may be positioned to a particular record within the data set either by a READ KEY or a DELETE KEY operation that specifies the key of the desired record. Thereafter, successive READ statements without the KEY option will access the following records in the data set sequentially. A subsequent READ statement without the KEY option causes the record with the next higher recorded key to be read (even if the keyed record has not been found).

The length of the recorded keys in an indexed data set is defined by the KEYLENGTH ENVIRONMENT option or the KEYLEN subparameter of the DD statement that defines the data set. If the length of a source key is greater than the specified length of the recorded keys, the source key is truncated on the right.

The effect of supplying a source key that is shorter than the recorded keys in the data set differs according to whether or not the GENKEY option is specified in the ENVIRONMENT attribute. In the absence of the GENKEY option, the source key is padded on the right with blanks to the length specified in the KEYLENGTH option of the ENVIRONMENT attribute, and the record with this padded key is read (if such a record exists). If the GENKEY option is specified, the source key is interpreted as a generic key, and the first record with a key in the class identified by this generic key is read. (For further details see, "GENKEY Option—Key Classification" on page 129.)

Direct Access

A direct file that is used to access an indexed data set may be opened with either the INPUT or the UPDATE attribute. All data transmission statements must include source keys; the DIRECT attribute implies the KEYED attribute.

A DIRECT UPDATE file can be used to retrieve, and delete, or replace records in an indexed data set according to the following conventions:

- **Retrieval:** If the DD statement that defined the data set included the subparameter OPTCD=L, dummy records are not made available by a READ statement (the KEY condition is raised).
- **Addition:** A WRITE statement that includes a unique key causes a record to be inserted into the data set. If the key is the same as the recorded key of a dummy record, the new record replaces the dummy record. If the key is the same as the recorded key of a record that is not marked as deleted, or if there is no space in the data set for the record, the KEY condition is raised.
- **Deletion:** The record specified by the source key in a DELETE statement is retrieved, marked as deleted, and rewritten into the data set. The effect of the DELETE statement is to insert the value (8)'1'B in the first byte of the data in a record. Deletion is possible only if OPTCD=L was specified in the DD statement that defined the data set when it was created. If the data set has F-format blocked records with RKP=0 or KEYLOC=1, or V-format records with RKP=4 or KEYLOC=1, records cannot be deleted. (The code (8)'1'B would overwrite the embedded keys.)

- **Replacement:** The record specified by a source key in a REWRITE statement is replaced by the new record. If the data set contains F-format blocked records, a record replaced with a REWRITE statement causes an implicit READ statement to be executed unless the previous I/O statement was a READ statement that obtained the record to be replaced. If the data set contains V-format records and the updated record has a length different from that of the record read, the whole of the remainder of the track will be removed, and may cause data to be moved to an overflow track.

Essential Information

To access an indexed data set, you must define it in one, two, or three DD statements; the DD statements must correspond with those used when the data set is created. The following paragraphs indicate the essential information you must include in each DD statement; Figure 76 summarizes this information.

If the data set is cataloged, you need supply only the following information in each DD statement:

- The name of the data set (DSNAME parameter). The operating system will locate the information that describes the data set in the system catalog and, if necessary, will request the operator to mount the volume that contains it.
- Confirmation that the data set exists (DISP parameter).
- Full DCB information for the first, or only, DD statement. Subsequent statements require only DSORG=IS to be coded.

If the data set is not cataloged, you must, in addition, specify the device that will process the data set and give the serial number of the volume that contains it (UNIT and VOLUME parameters).

REORGANIZING AN INDEXED DATA SET

It is necessary to reorganize an indexed data set periodically because the addition of records to the data set results in an increasing number of records in the overflow area. Therefore, even if the overflow area does not eventually become full, the average time required for the direct retrieval of a record will increase. The frequency of reorganization depends on how often the data set is updated, on how much storage is available in the data set, and on your timing requirements.

Reorganizing the data set also eliminates records that are marked as "deleted," but are still present within the data set.

There are two ways to reorganize an indexed data set:

- Read the data set into an area of main storage or onto a temporary consecutive data set, and then re-create it in the original area of auxiliary storage.
- Read the data set sequentially and write it into a new area of auxiliary storage; you can then release the original auxiliary storage.

Parameters of DD Statement		
When Required	**What You Must State**	**Parameters**
Always	Name of data set	DSNAME=
	Disposition of data set	DISP=
	Data control block information	DCB=
If data set not cataloged	Input device	UNIT= or VOLUME=REF=
	Volume serial number	VOLUME=SER=

Figure 76. Accessing an Indexed Data Set: Essential Parameters of DD statement

EXAMPLES OF INDEXED DATA SETS

The creation of a simple indexed data set is illustrated in Figure 77 on page 187. The data set contains a telephone directory, using the subscribers' names as keys to the telephone numbers.

```
//EX8#19 JOB
//STEP1 EXEC PLIXCLG
//PLI.SYSIN  DD  *
 TELNOS: PROC OPTIONS(MAIN);
         DCL DIREC FILE RECORD SEQUENTIAL KEYED ENV(INDEXED),
             CARD CHAR(80),
             NAME CHAR(20) DEF CARD,
             NUMBER CHAR(3) DEF CARD POS(21),
             IOFIELD CHAR(3);
         ON ENDFILE(SYSIN) GO TO FINISH;
         OPEN FILE(DIREC) OUTPUT;
 NEXTIN: GET FILE(SYSIN) EDIT(CARD)(A(80));
         PUT FILE(SYSPRINT) SKIP EDIT (CARD) (A);
         IOFIELD=NUMBER;
         WRITE FILE(DIREC) FROM(IOFIELD) KEYFROM(NAME);
         GO TO NEXTIN;
 FINISH: CLOSE FILE(DIREC);
   END TELNOS;
/*
//GO.DIREC DD DSN=HPU8.TELNO(INDEX),UNIT=SYSDA,VOL=SER=nnnnnn,
//            DCB=(RECFM=F,BLKSIZE=3,DSORG=IS,KEYLEN=20,OPTCD=LIY,
//            CYLOFL=2),SPACE=(CYL,1),DISP=(NEW,CATLG)
//         DD DSN=HPU8.TELNO(PRIME),UNIT=SYSDA,VOL=SER=nnnnnn,
//            DISP=(NEW,KEEP),DCB=DSORG=IS,SPACE=(CYL,1)
//         DD DSN=HPU8.TELNO(OVFLOW),UNIT=SYSDA,VOL=SER=nnnnnn,
//            DISP=(NEW,KEEP),DCB=DSORG=IS,SPACE=(CYL,1)
//GO.SYSIN DD *
ACTION,G.           162
BAKER,R.            152
BRAMLEY,O.H.        248
CHEESEMAN,D.        141
CORY,G.             336
ELLIOTT,D.          875
FIGGINS,S.          413
HARVEY,C.D.W.       205
HASTINGS,G.M.       391
KENDALL,J.G.        294
LANCASTER,W.R.      624
MILES,R.            233
NEWMAN,M.W.         450
PITT,W.H.           515
ROLF,D.E.           114
SHEERS,C.D.         241
SUTCLIFFE,M.        472
TAYLOR,G.C.         407
WILTON,L.W.         404
WINSTONE,E.M.       307
/*
```

Figure 77. Creating an Indexed Data Set

The program in Figure 78 on page 188 updates this data set and prints out its new contents. The input data includes the following codes to indicate the operations required:

A: Add a new record
C: Change an existing record
D: Delete an existing record

```
//EX8#20 JOB
//STEP1 EXEC PLIXCLG
//PLI.SYSIN  DD  *
 DIRUPDT: PROC OPTIONS(MAIN);
          DCL DIREC FILE RECORD KEYED ENV(INDEXED),
              NUMBER CHAR(3),NAME CHAR(20),CODE CHAR(2),ONCODE BUILTIN;
          ON ENDFILE(SYSIN) GO TO PRINT;
          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;
 NEXT:    GET FILE(SYSIN) EDIT(NAME,NUMBER,CODE)
            (COLUMN(1),A(20),A(3),A(1));
          PUT FILE(SYSPRINT) SKIP EDIT (' ',NAME,'#',NUMBER,' ',CODE)
             (A(1),A(20),A(1),A(3),A(1),A(1);
          IF CODE='A' THEN
              WRITE FILE(DIREC) FROM(NUMBER) KEYFROM(NAME);
          ELSE IF CODE='C' THEN
              REWRITE FILE(DIREC) FROM(NUMBER) KEY(NAME);
          ELSE IF CODE='D' THEN
              DELETE FILE(DIREC) KEY(NAME);
          ELSE PUT FILE(SYSPRINT) SKIP
               EDIT('INVALID CODE:',NAME)(A(15),A);
          GO TO NEXT;
 PRINT:   CLOSE FILE(DIREC);
          PUT FILE(SYSPRINT) PAGE;
          OPEN FILE(DIREC) SEQUENTIAL INPUT;
          ON ENDFILE(DIREC) GO TO FINISH;
 NEXTIN:  READ FILE(DIREC) INTO(NUMBER) KEYTO(NAME);
          PUT FILE(SYSPRINT) SKIP EDIT(NAME,NUMBER)(A);
          GO TO NEXTIN;
 FINISH:  CLOSE FILE(DIREC);        END DIRUPDT;
/*
//GO.DIREC DD DSN=MY.TELNO(INDEX),DISP=(OLD,KEEP),UNIT=SYSDA,
//      VOL=SER=nnnnnn
//           DD DSN=MY.TELNO(PRIME),DISP=(OLD,KEEP),UNIT=SYSDA,
//      VOL=SER=nnnnnn
//           DD DSN=MY.TELNO(OVFLOW),DISP=(OLD,KEEP),UNIT=SYSDA,
//      VOL=SER=nnnnnn
//GO.SYSIN DD *
NEWMAN,M.W.          516C
GOODFELLOW,D.T.      889A
MILES,R.                D
HARVEY,C.D.W.        209A
BARTLETT,S.G.        183A
CORY,G.                 D
READ,K.M.            001A
PITT,W.H.
ROLF,D.E.               D
ELLIOTT,D.           291C
HASTINS,G.M.            D
BRAMLEY,O.H.         439
/*
```

Figure 78. Updating an Indexed Data Set

REGIONAL DATA SETS

This section describes regional data set organization, the data transmission statements, and the ENVIRONMENT options that define regional data sets. It then describes, for each type of regional organization in turn, how to create and access regional data sets.

REGIONAL ORGANIZATION

A data set with regional organization is divided into regions, each of which is identified by a region number, and each of which may 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 may be accessed by specifying its region number, and perhaps a key, in a data transmission statement.

Regional data sets are confined to direct-access devices.

Regional organization of a data set permits you to control the physical placement of records in the data set, and to optimize the access time for a particular application. Such optimization is not available with consecutive or indexed organization, in which successive records are written either in strict physical sequence or in logical sequence depending on ascending key values; neither of these methods takes full advantage of the characteristics of direct-access storage devices.

A regional data set can be created in a manner similar to a consecutive or indexed data set, records being presented in the order of ascending region numbers; alternatively, direct-access can be used, in which records can be presented in random sequence and inserted directly into preformatted regions. Once a regional data set has been created, it can be accessed by a file with the attributes SEQUENTIAL or DIRECT as well as INPUT or UPDATE. Neither a region number nor a key need be specified if the data set is associated with a SEQUENTIAL INPUT or SEQUENTIAL UPDATE file. When the file has the DIRECT attribute, records can be retrieved, added, deleted, and replaced at random.

Records within a regional data set are either actual records containing valid data or dummy records. The nature of the dummy records depends on the type of regional organization; the three types of regional organization are described below.

The major advantage of regional organization over other types of data set organization is that it allows you to control the relative placement of records; by judicious programming, you can optimize record access in terms of device capabilities and the requirements of particular applications.

Direct access of regional data sets is quicker than that of indexed data sets, but they have the disadvantage that sequential processing may 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.

Figure 79 lists the data transmission statements and options that can be used to create and access a regional data set.

File declaration[1]	Valid statements[2] with options that must appear	Other options that can also be used
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); READ FILE(file-reference) SET(pointer-reference); READ FILE(file-reference) IGNORE(expression);	KEYTO(reference) KEYTO(reference)
SEQUENTIAL INPUT UNBUFFERED	READ FILE(file-reference) INTO(reference); READ FILE(file-reference) IGNORE(expression);	EVENT(event-reference) and/or KEYTO(reference) 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);	KEYTO(reference) KEYTO(reference) FROM(reference)
SEQUENTIAL UPDATE UNBUFFERED	READ FILE(file-reference) INTO(reference); READ FILE(file-reference) IGNORE(expression); REWRITE FILE(file-reference) FROM(reference);	EVENT(event-reference) and/or KEYTO(reference) EVENT(event-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)

Figure 79 (Part 1 of 2). Statements and Options Permitted for Creating and Accessing Regional Data Sets

File declaration[1]	Valid statements[2] with options that must appear	Other options that can also be used
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)
DIRECT INPUT EXCLUSIVE	READ FILE(file-reference) INTO(reference) KEY(expression);	EVENT(event-reference) and/or NOLOCK
DIRECT UPDATE EXCLUSIVE	READ FILE(file-reference) INTO(reference) KEY(expression);	EVENT(event-reference) and/or NOLOCK
	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)
	UNLOCK FILE(file-reference) KEY(expression);	

Figure 79 (Part 2 of 2). Statements and Options Permitted for Creating and Accessing Regional Data Sets

Notes to Figure 79:

1 The complete file declaration would include the attributes FILE, RECORD, and ENVIRONMENT; if any of the options KEY, KEYFROM, or KEYTO is used, 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);

DEFINING A REGIONAL DATA SET

A sequential regional data set is defined by a file declaration with the following attributes:

```
DCL filename FILE RECORD
             INPUT | OUTPUT | UPDATE
             SEQUENTIAL
             BUFFERED | UNBUFFERED
            [KEYED]
             ENVIRONMENT(options);
```

A direct regional data set is defined by a file declaration with the following attributes:

```
DCL filename FILE RECORD
             INPUT | OUTPUT | UPDATE
             DIRECT
             UNBUFFERED
             KEYED
            [EXCLUSIVE]
             ENVIRONMENT(options);
```

Default file attributes are shown in Figure 45 on page 123. The file attributes are described in the OS and DOS PL/I Language Reference Manual. Options of the ENVIRONMENT attribute are discussed below.

ENVIRONMENT OPTIONS FOR REGIONAL DATA SETS

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
BUFFERS(n)
KEYLENGTH(n)
NCP(n)
TRKOFL
```

All the options except REGIONAL are described in Chapter 4, "Data Sets and Files" on page 100, while REGIONAL is described below.

REGIONAL Option

The REGIONAL option defines a file with regional organization.

Syntax

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

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).

Although REGIONAL(1) data sets have no recorded keys, REGIONAL(1) DIRECT INPUT or UPDATE files can be used to process data sets that do have recorded keys. In particular, REGIONAL(2) and REGIONAL(3) data sets can be accessed by a file declared with REGIONAL(1) organization.

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 files that are 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 below.

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.

REGIONAL(3) organization is similar to REGIONAL(2) in that records contain recorded keys, but differs in that a region for REGIONAL(3) corresponds to a track and not a record position.

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 may 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 may 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 at the end of this section illustrate typical applications of all three types of regional organization.

KEYS

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 a record in a regional data set is accessed, the source key gives a region number, and may also give a recorded key.

The length of the recorded keys in a regional data set is specified by the KEYLENGTH option of the ENVIRONMENT attribute, or the KEYLEN subparameter on the DD statement. Unlike the keys for indexed data sets, recorded keys in a regional data set are never embedded within the record.

REGIONAL(1) ORGANIZATION

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 may 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. Embedded blanks are not permitted in the number; the first embedded blank, if any, terminates the region number. If more than 8 characters appear in the source key, only the rightmost 8 are used as the region number; if there are fewer than 8 characters, blanks (interpreted as zeros) are inserted on the left.

Dummy Records

Records in a REGIONAL(1) data set are either actual records containing valid data or dummy records. A dummy record in a REGIONAL(1) data set is identified by the constant (8)'1'B in its first byte. Although such dummy records are inserted in the data set either when it is created or when a record is deleted, they are not ignored when the data set is read; the PL/I program must be prepared to recognize them. Dummy records can be replaced by valid data. Note that if you insert (8)'1'B in the first byte, the record will be lost if the file is copied onto a data set whose dummy records are not retrieved.

Creating a REGIONAL(1) Data Set

A REGIONAL(1) data set can be created either sequentially or by direct-access. Figure 79 on page 189 shows the statements and options for creating a regional data set.

When a SEQUENTIAL OUTPUT file is used to create the data set, the opening of the file causes all tracks on the data set to be cleared, and a capacity record to be written at the beginning of each track to record the amount of space available on that track. Records must be presented in ascending order of region numbers; any region that is omitted from the sequence is filled with a dummy record. If there is an error in the sequence, or if a duplicate key is presented, the KEY condition is raised. When the file is closed, any space remaining at the end of the current extent is filled with dummy records.

If a data set is created 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 may raise the ERROR condition.

If a DIRECT OUTPUT file is used to create the data set, the whole of the primary extent allocated to the data set is filled with dummy records when the file is opened. Records can be presented in random order; if a duplicate key is presented, the KEY condition is raised.

For sequential creation, the data set can have up to 15 extents, which may be on more than one volume. For direct creation, the data set can have only one extent, and can therefore reside on only one volume.

Accessing a REGIONAL(1) Data Set

Once a REGIONAL(1) data set has been created, the file that accesses it can be opened for SEQUENTIAL INPUT or UPDATE, or for DIRECT INPUT or UPDATE. It can be opened for OUTPUT only if the existing data set is to be overwritten. Figure 79 on page 189 shows the statements and options for accessing a regional data set.

SEQUENTIAL ACCESS: A SEQUENTIAL file that is used to process a REGIONAL(1) data set may be opened with either the INPUT or UPDATE attribute. The data transmission statements must not include the KEY option; but the file may have the KEYED attribute, since the KEYTO option can be used. 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 the PL/I program should be prepared to recognize dummy records.

Using sequential input with a REGIONAL(1) data set, you can read all the records in ascending region-number sequence, and in sequential update you can read and may 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 "Consecutive Data Sets" on page 149.

DIRECT ACCESS: A DIRECT file that is used to process a REGIONAL(1) data set may be opened with either the INPUT or the UPDATE attribute. All data transmission statements must include source keys; the DIRECT attribute implies the KEYED attribute.

A DIRECT UPDATE file can be used 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. The program must be prepared to 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 specified by the source key in a DELETE statement is converted to a dummy record.
- **Replacement:** The record specified by the source key in a REWRITE statement, whether dummy or actual, is replaced.

REGIONAL(2) ORGANIZATION

In a REGIONAL(2) data set, 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 is supplied in the source key of the WRITE statement that adds the record to the data set.

When a record is added 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 that contains 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. Unless it is limited

by the LIMCT subparameter of the DD statement that defines the data set, the search for a record or for space to add a record continues right through to the end of the data set and then from the beginning until the whole of the data set has been covered. The closer a record is to the specified region, the more quickly it can be accessed.

Source Keys

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 (although the actual number of records allowed may 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. The region specification can include only the characters 0 through 9 and the blank character; leading blanks are interpreted as zeros. Embedded blanks are not permitted in the number; the first embedded 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 may overlap or be distinct from the region number, from which it can be separated by other, nonsignificant, characters. The length of the comparison key is specified by either the KEYLEN subparameter of the DD statement for the data set or the KEYLENGTH option of the ENVIRONMENT attribute.If the source key is shorter than the specified key length, 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; alternatively, part of the source key may not be used. The length of the comparison key is always equal to KEYLENGTH or KEYLEN; if the source key is longer than KEYLEN+8, the characters in the source key between the comparison key and the region number are ignored.

When generating the key, the rules for arithmetic to character string conversion should be considered. For example, the following group would be in error:

```
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.

Consider the following examples of source keys (the character "b" represents a blank):

```
KEY ('JOHNbDOEbbbbbb12363251')
```

The rightmost 8 characters make up the region specification, the relative number of the record. Assume that the associated DD statement has the subparameter KEYLEN=14. In retrieving a record, the search begins with the beginning of the track that contains the region number 12363251, until the record is found having the recorded key of JOHNbDOEbbbbbb.

If the subparameter were KEYLEN=22, the search still would begin 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 'JOHNbDOEbbbbbb12363251'.

```
KEY('JOHNbDOEbbbbbbDIVISIONb423bbbb34627')
```

In this example, the rightmost 8 characters contain leading blanks, which are interpreted as zeros. The search begins at region number 00034627. If KEYLEN=14 is specified, the characters DIVISIONb423b will be ignored.

Assume that COUNTER is declared FIXED BINARY(21) and NAME is declared CHARACTER(15). The key might be specified 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—3 blanks followed by 8 decimal digits.) The value of the rightmost 8 characters of the converted string is taken to be the region specification. Then if the keylength specification is KEYLEN=15, the value of NAME is taken to be the comparison specification.

Dummy Records

A REGIONAL(2) data set can contain dummy records. A dummy record consists of a dummy key and dummy data. A dummy key is identified by the constant (8)'1'B in its first byte. The first byte of the data contains the sequence number of the record on the track.

Dummy records can be replaced by valid data. They are inserted either when the data set is created or when a record is deleted, and they are ignored when the data set is read.

Creating a REGIONAL(2) Data Set

A REGIONAL(2) data set can be created either sequentially or by direct-access. In either case, when the file associated with the data set is opened, the data set is initialized with capacity records specifying the amount of space available on each track. Figure 79 on page 189 shows the statements and options for creating a regional data set.

When a SEQUENTIAL OUTPUT file is used to create the data set, records must be presented in ascending order of region numbers; any region that is omitted from the sequence is filled with a dummy record. If there is an error in the sequence, including an attempt to place more than one record in the same region, the KEY condition is raised. When the file is closed, any space remaining at the end of the current extent is filled with dummy records.

If a data set is created 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 may raise the ERROR condition.

If a DIRECT OUTPUT file is used to create the current extent of a data set, the whole of the primary extent allocated to the data set is filled with dummy records when the file is opened. Records can be presented in random order, and no condition is raised by duplicate keys. Each record is substituted for the first dummy record on the track that contains the region specified in the source key; if there are no dummy records on the track, the record is substituted for the first dummy record encountered on a subsequent track, unless the LIMCT subparameter specifies that the search cannot reach beyond this track. (Note that it is possible to place records with identical recorded keys in the data set).

For sequential creation, the data set can have up to 15 extents, which may be on more than one volume. For direct creation, the data set can have only one extent, and can therefore reside on only one volume.

Accessing a REGIONAL(2) Data Set

Once a REGIONAL(2) data set has been created, the file that accesses it can be opened for SEQUENTIAL INPUT or UPDATE, or for DIRECT INPUT or UPDATE. It cannot be opened for OUTPUT. Figure 79 on page 189 shows the statements and options for accessing a regional data set.

Sequential Access

A SEQUENTIAL file that is used to process a REGIONAL(2) data set may be opened with either the INPUT or the UPDATE attribute. The data transmission statements must not include the KEY option, but the file may have the KEYED attribute since the KEYTO option can be used. The KEYTO option specifies 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 are specified in the KEYLEN subparameter, the value returned (the recorded key) is extended on the right with blanks; if it has fewer characters than specified by KEYLEN, the value returned is truncated on the right.

Sequential access is in the physical order in which the records exist on the data set, 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) data set are identical with those for a CONSECUTIVE data set (described above).

DIRECT ACCESS: A DIRECT file that is used to process a REGIONAL(2) data set may be opened with either the INPUT or the UPDATE attribute. All data transmission statements must include source keys; the DIRECT attribute implies the KEYED attribute. The search for each record is commenced at the start of the track containing the region number indicated by the key.

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:** A WRITE statement substitutes the new record for the first dummy record on the track containing the region specified by the source key. If there are no dummy records on this track, and an extended search is permitted by the LIMCT subparameter, the new record replaces the first dummy record encountered during the search.
- **Deletion:** The record specified by the source key in a DELETE statement is converted to a dummy record.
- **Replacement:** The record specified by the source key in a REWRITE statement must exist; a REWRITE statement cannot be used to replace a dummy record. If it does not exist, the KEY condition is raised.

REGIONAL(3) ORGANIZATION

A REGIONAL(3) data set differs from a REGIONAL(2) data set (described above) only in the following respects:

- Each region number identifies a track on the direct-access device that contains the data set; the region number should not exceed 32767. A region in excess of 32767 is treated as modulo 32768; for example, 32778 is treated as 10.
- A region can contain one or more records, or a segment of a VS-format record.
- The data set can contain F-format, V-format, VS-format, or U-format records. Dummy records can be created, but a data set that has V-format, VS-format, or U-format records is not preformatted with dummy records because the lengths of records cannot be known until they are written; however, all tracks in the primary extent are cleared and the operating system maintains a capacity record at the beginning of each track, in which it records the amount of space available on that track.

Source keys for a REGIONAL(3) data set are interpreted exactly as those for a REGIONAL(2) data set are, and the search for a record or space to add a record is conducted in a similar manner.

Dummy Records

Dummy records for REGIONAL(3) data sets with F-format records are identical with those for REGIONAL(2) data sets.

V-format, VS-format, and U-format dummy records are identified by the fact that they have dummy recorded keys ((8)'1'B in the first byte). The 4 control bytes in each V-format and VS-format dummy record are retained, but otherwise the contents of V-format, VS-format, and U-format dummy records are undefined. V-format, VS-format, and U-format records are converted to dummy records only when a record is deleted; they cannot be reconverted to valid records.

Creating a REGIONAL(3) Data Set

A REGIONAL(3) data set can be created either sequentially or by direct-access. In either case, when the file associated with the data set is opened, the data set is initialized with capacity records specifying the amount of space available on each track. Figure 79 on page 189 shows the statements and options for creating a regional data set.

When a SEQUENTIAL OUTPUT file is used to create the data set, records must be presented in ascending order of region numbers, but the same region number can be specified for successive records. For F-format records, any record that is omitted from the sequence is filled with a dummy record. If there is 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 a data set is created 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 may raise the ERROR condition.

If a DIRECT OUTPUT file is used to create the data set, the whole of the primary extent allocated to the data set is initialized when the data set 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. Records can be presented 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 in the region (track) specified on the source key; if there are no dummy records on the track, and an extended search is permitted by the LIMCT subparameter, the record is substituted for the first dummy record encountered during the search. 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; otherwise, if an extended search is permitted, the new record is inserted in the next available space.

Note that for spanned records, space may be required for overflow onto subsequent tracks.

For sequential creation, the data set can have up to 15 extents, which may be on more than one volume. For direct creation, the data set can have only one extent, and can therefore reside on only one volume.

Accessing a REGIONAL(3) Data Set

Once a REGIONAL(3) data set has been created, the file that accesses it can be opened for SEQUENTIAL INPUT or UPDATE, or for DIRECT INPUT or UPDATE. It can only be opened for OUTPUT if the entire existing data set is to be deleted and replaced. Figure 79 on page 189 shows the statements and options for accessing a regional data set.

SEQUENTIAL ACCESS: A SEQUENTIAL file that is used to access a REGIONAL(3) data set may be opened with either the INPUT or UPDATE attribute. The data transmission statements must not include the KEY option, but the file may have the KEYED attribute since the KEYTO option can be used. The KEYTO option specifies 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 are specified in the KEYLEN subparameter, value returned (the recorded key) is extended on the right with blanks; if it has fewer characters than specified by KEYLEN, the value returned is truncated on the right.

Sequential access is in the order of ascending relative tracks. 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(3) data set are identical with those for a CONSECUTIVE data set (described above).

DIRECT ACCESS: A DIRECT file that is used to process a REGIONAL(3) data set may be opened with either the INPUT or the UPDATE attribute. All data transmission statements must include source keys; 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 data set with F-format records, a WRITE statement substitutes the new record for a dummy record in the region (track) specified by the source key. If there are no dummy records on the specified track, and an extended search is permitted by the LIMCT subparameter, the new

record replaces the first dummy record encountered during the search. 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 on the specified track if space is available; otherwise, if an extended search is permitted, the new record is inserted in the next available space.

- **Deletion:** A record specified by the source key in a DELETE statement is converted to a dummy record. The space formerly occupied by an F-format record can be re-used; space formerly occupied by V-format, VS-format, or U-format records is not available for re-use.

- **Replacement:** The record specified by the source key in a REWRITE statement must exist; a REWRITE statement cannot be used 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.

ESSENTIAL INFORMATION FOR CREATING AND ACCESSING REGIONAL DATA SETS

To create a regional data set, you must give the operating system certain information, either in your PL/I program or in the DD statement that defines the data set. The following paragraphs indicate the essential information, and discuss some of the optional information you may supply.

You must supply the following information when creating a regional data set:

- Device that will write your data set (UNIT or VOLUME parameter of DD statement).

- Block size: You can specify the block size either in your PL/I program (in the BLKSIZE option of the ENVIRONMENT attribute) or in the DD statement (BLKSIZE subparameter). If you do not specify a record length, unblocked records are the default and the record length is determined from the block size.

If you want to keep a data set (that is, you do not want the operating system to delete it at the end of your job), the DD statement must name the data set and indicate how it is to be disposed of (DSNAME and DISP parameters). The DISP parameter alone will suffice if you want to use the data set in a later step but will not need it after the end of your job.

If you want your data set stored on a particular direct-access device, you must indicate the volume serial number in the DD statement (SER or REF subparameter of VOLUME parameter). If you do not supply a serial number for a data set that you want to keep, the operating system will allocate one, inform the operator, and print the number on your program listing. All the essential parameters required in a DD statement for the creation of a regional data set are summarized in Figure 80 on page 202; and Figure 81 on page 203 lists the DCB subparameters needed. See your JCL manual for a description of the DCB subparameters.

You cannot place a regional data set on a system output (SYSOUT) device.

In the DCB parameter, you must always specify the data set organization as direct by coding DSORG=DA. You cannot specify the DUMMY or DSN=NULLFILE parameters in a DD statement for a regional data set. For REGIONAL(2) and REGIONAL(3), you must also specify the length of the recorded key (KEYLEN) unless it is specified in the ENVIRONMENT attribute; see "Source Keys" on page 196 for a description of how the recorded key is derived from the source key supplied in the KEYFROM option.

Parameters of DD Statement

When Required	What You Must State	Parameters
Always	Output device[1]	UNIT= or VOLUME=REF
	Storage space required[2]	SPACE=
	Data control block information: see Figure 81 on page 203.	DCB=
Data set to be used in another job step but not required in another job	Disposition	DISP=
Data set to be kept after end of job	Disposition	DISP=
	Name of data set	DSNAME=
Data set to be on particular volume	Volume serial number	VOLUME=SER= or VOLUME=REF=

[1] Regional data sets are confined to direct-access devices.

[2] For sequential access, the data set can have up to 15 extents, which may be on more than one volume.
For creation with DIRECT access, the data set can have only one extent.

Figure 80. Creating a Regional Data Set: Essential Parameters of DD Statement

For REGIONAL(2) and REGIONAL(3), if you want to restrict the search for space to add a new record, or the search for an existing record, to a limited number of tracks beyond the track that contains the specified region, use the LIMCT subparameter of the DCB parameter. If you omit this parameter, the search will continue to the end of the data set, and then from the beginning of the data set back to the starting point.

To access a regional data set, you must identify it to the operating system in a DD statement. The following paragraphs indicate the minimum information you must include in the DD statement; this information is summarized in Figure 82 on page 203.

If the data set is cataloged, you need supply only the following information in your DD statement:

- The name of the data set (DSNAME parameter). The operating system will locate the information that describes the data set in the system catalog and, if necessary, will request the operator to mount the volume that contains it.
- Confirmation that the data set exists (DISP parameter).

If the data set is not cataloged, you must, in addition, specify the device that will read the data set and give the serial number of the volume that contains the data set (UNIT and VOLUME parameters).

Unlike indexed data sets, regional data sets do not require the subparameter OPTCD=L in the DD statement.

When opening a multiple volume regional data set for sequential update, the ENDFILE condition is raised at the end of the first volume.

DCB Subparameters

When required	To Specify	Subparameters
These are always required	Record format[1]	RECFM=F or RECFM=V[2] REGIONAL(3) only or RECFM=U REGIONAL(3) only
	Block size[1]	BLKSIZE=
	Data set organization	DSORG=DA
	Key length (REGIONAL(2) and (3) only)[1]	KEYLEN=
These are optional	Limited search for a record or space to add a record (REGIONAL(2) and (3) only)	LIMCT=
	Number of data management buffers[1]	BUFNO=

[1] Alternatively, can be specified in ENVIRONMENT attribute.

[2] RECFM=VS must be specified in the ENVIRONMENT attribute for sequential input or update.

Figure 81. DCB Subparameters for a Regional Data Set

Parameters of DD Statement

When Required	What You Must State	Parameters
Always	Name of data set	DSNAME=
	Disposition of data set	DISP=
If data set not cataloged	Input device	UNIT= or VOLUME=REF=
	Volume serial number	VOLUME=SER=

Figure 82. Accessing a Regional Data Set: Essential Parameters of DD Statement

EXAMPLES OF REGIONAL DATA SETS

REGIONAL(1) Data Sets

Creating a REGIONAL(1) data set is illustrated in Figure 83 on page 206. 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.

Updating a REGIONAL(1) data set is illustrated in Figure 84 on page 207. Like the program in Figure 78 on page 188, this program updates the data set and lists its contents. Before each new or updated record is written, the existing record in the region is tested to ensure that it is a dummy; this is necessary because a WRITE statement can overwrite an existing record in a REGIONAL(1) data set even if it is not a dummy. Similarly, during the sequential reading and printing of the contents of the data set, each record is tested and dummy records are not printed.

REGIONAL(2) Data Sets

The use of REGIONAL(2) data sets is illustrated in Figure 85 on page 208, Figure 86 on page 209, and Figure 87 on page 210. The programs in these figures perform the same functions as those given for REGIONAL(3), with which they can be compared.

The programs depict a library processing scheme, in which loans of books are recorded and reminders are issued for overdue books. Two data sets, &&STOCK and &&LOANS are used. &&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.) &&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 as a region number in &&LOANS; the reader and book numbers are concatenated to form the recorded keys.

In Figure 85 on page 208, the data sets &&STOCK and &&LOANS are created. The file LOANS, which is used to create the data set &&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. Direct creation is also used for the data set &&STOCK because, even if the input data is presented in ascending reference number order, identical region numbers might be derived from successive reference numbers.

Updating of the data set &&LOANS is illustrated in Figure 86 on page 209. Each item of input data, read from a punched card, 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. To make the example self-contained it is assumed that several days' entries will be presented at one time and that daily entries are separated by a record starting with asterisks. Thus the deletion function can be tested.

The program in Figure 87 on page 210 uses a sequential update file (LOANS) to process the records in the data set &&LOANS, and a direct input file (STOCK) to obtain the book description from the data set &&STOCK for use in a reminder note. Each record from &&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.

REGIONAL(3) Data Sets

The use of REGIONAL(3) data sets, illustrated in Figure 88 on page 211, Figure 89 on page 212, and Figure 90 on page 213. is similar to the REGIONAL(2) figures, above; only the important differences are discussed here.

In Figure 88 on page 211, the data set &&STOCK is created sequentially; duplicate region numbers are acceptable, because each region can contain more than one record.

In Figure 89 on page 212, the region number for the data set &&LOANS is obtained simply by testing the reader number.

Figure 90 on page 213 is very much like Figure 87 on page 210, the REGIONAL(2) example.

```
//EX9    JOB
//STEP1  EXEC PLIXCLG,PARM.PLI='NOP,MAR(1,72)',PARM.LKED='LIST'
//PLI.SYSIN  DD  *
 CRR1:   PROC OPTIONS(MAIN);
  /*  CREATING A REGIONAL(1) DATA SET  -  PHONE DIRECTORY     */

   DCL NOS FILE RECORD OUTPUT DIRECT KEYED ENV(REGIONAL(1));
   DCL  SYSIN FILE INPUT RECORD;
   DCL  SYSIN_REC BIT(1) INIT('1'B);
   DCL 1   CARD,
        2   NAME   CHAR(20),
        2   NUMBER CHAR( 2),
        2   CARD_1 CHAR(58);
   DCL IOFIELD CHAR(20);

      ON ENDFILE (SYSIN) SYSIN_REC = '0'B;
      OPEN FILE(NOS);
      READ FILE(SYSIN) INTO(CARD);

      DO WHILE(SYSIN_REC);
         IOFIELD = NAME;
         WRITE FILE(NOS) FROM(IOFIELD) KEYFROM(NUMBER);
         PUT FILE(SYSPRINT) SKIP EDIT (CARD) (A);
         READ FILE(SYSIN) INTO(CARD);
      END;

      CLOSE FILE(NOS);
  END CRR1;
/*
//GO.SYSLMOD DD DSN=&&GOSET,DISP=(OLD,DELETE)
//GO.NOS     DD DSN=NOS,UNIT=SYSDA,SPACE=(20,100),
//           DCB=(RECFM=F,BLKSIZE=20,DSORG=DA),DISP=(NEW,KEEP)
//GO.SYSIN DD *
ACTION,G.            12
BAKER,R.             13
BRAMLEY,O.H.         28
CHEESNAME,L.         11
CORY,G.              36
ELLIOTT,D.           85
FIGGINS,E.S.         43
HARVEY,C.D.W.        25
HASTINGS,G.M.        31
KENDALL,J.G.         24
LANCASTER,W.R.       64
MILES,R.             23
NEWMAN,M.W.          40
PITT,W.H.            55
ROLF,D.E.            14
SHEERS,C.D.          21
SURCLIFFE,M.         42
TAYLOR,G.C.          47
WILTON,L.W.          44
WINSTONE,E.M.        37
/*
```

Figure 83. Creating a REGIONAL(1) Data Set

```
//EX10    JOB
//STEP2   EXEC  PLIXCLG,PARM.PLI='NOP,MAR(1,72)',PARM.LKED='LIST'
//PLI.SYSIN  DD  *
 ACR1: PROC OPTIONS(MAIN);
   /*  UPDATING A REGIONAL(1) DATA SET  -  PHONE DIRECTORY      */
    DCL NOS FILE RECORD  KEYED ENV(REGIONAL(1));
    DCL  SYSIN FILE INPUT RECORD;
    DCL  (SYSIN_REC,NOS_REC) BIT(1) INIT('1'B);
    DCL 1   CARD,
         2   NAME   CHAR(20),
         2   (NEWNO,OLDNO)  CHAR( 2),
         2   CARD_1 CHAR( 1),
         2   CODE   CHAR( 1),
         2   CARD_2 CHAR(54);
    DCL IOFIELD CHAR(20);
    DCL BYTE    CHAR(1) DEF IOFIELD;

   ON ENDFILE(SYSIN) SYSIN_REC = '0'B;
   OPEN FILE (NOS) DIRECT UPDATE;
   READ FILE(SYSIN) INTO(CARD);

   DO WHILE(SYSIN_REC);
      SELECT(CODE);
         WHEN('A','C') DO;
             IF CODE = 'C' THEN
                DELETE FILE(NOS) KEY(OLDNO);
             READ FILE(NOS) KEY(NEWNO) INTO(IOFIELD);
             IF UNSPEC(BYTE) = (8)'1'B
                THEN WRITE FILE(NOS) KEYFROM(NEWNO) FROM(NAME);
             ELSE PUT FILE(SYSPRINT) SKIP LIST ('DUPLICATE:',NAME);
         END;
         WHEN('D') DELETE FILE(NOS) KEY(OLDNO);
      OTHERWISE PUT FILE(SYSPRINT) SKIP LIST ('INVALID CODE:',NAME);
      END;
      READ FILE(SYSIN) INTO(CARD);
   END;

   CLOSE FILE(SYSIN),FILE(NOS);
   PUT FILE(SYSPRINT) PAGE;
   OPEN FILE(NOS) SEQUENTIAL INPUT;
   ON ENDFILE(NOS) NOS_REC = '0'B;
   READ FILE(NOS) INTO(IOFIELD) KEYTO(NEWNO);
   DO WHILE(NOS_REC);
      IF UNSPEC(BYTE)    = (8)'1'B
         THEN PUT FILE(SYSPRINT) SKIP EDIT (NEWNO,IOFIELD)(A(2),X(3),A);
      PUT FILE(SYSPRINT) SKIP EDIT (IOFIELD) (A);
      READ FILE(NOS) INTO(IOFIELD) KEYTO(NEWNO);
   END;
  CLOSE FILE(NOS);
  END ACR1;
/*
//GO.NOS   DD DSN=NOS,DISP=(OLD,DELETE),UNIT=SYSDA,VOL=SER=nnnnnn
//GO.SYSIN DD *
NEWMAN,M.W.          5640 C
GOODFELLOW,D.T.      89   A
MILES,R.               23 D
HARVEY,C.D.W.        29   A
BARTLETT,S.G.        13   A
CORY,G.                36 D
READ,K.M.            01   A
PITT,W.H.              55
ROLF,D.F.              14 D
ELLIOTT,D.           4285 C
HASTINGS,G.M.          31 D
BRAMLEY,O.H.         4928 C
```

Figure 84. Updating a REGIONAL(1) Data Set

```
//EX11    JOB
//STEP1   EXEC PLIXCLG,PARM.PLI='NOP',PARM.LKED='LIST'
//PLI.SYSIN  DD  *
*PROCESS MAR(1,72);
CRR2: PROC OPTIONS(MAIN);
 /*  CREATING A REGIONAL(2) DATA SET - LIBRARY LOANS       */

   DCL (LOANS,STOCK) FILE RECORD KEYED ENV(REGIONAL(2));
   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);

    /* INITIALIZE (FORMAT) LOANS DATA SET    */
      OPEN FILE(LOANS) DIRECT OUTPUT;
      CLOSE FILE(LOANS);
      ON ENDFILE(SYSIN) GO TO FINISH;
      OPEN FILE(STOCK) DIRECT OUTPUT;

 NEXT: GET FILE(SYSIN) SKIP LIST(NUMBER,BOOK);
       INTER = (NUMBER-1000)/9;        /* REGIONS 0 TO 999 */
       REGION = INTER;
       WRITE FILE(STOCK) FROM (BOOK) KEYFROM(NUMBER||REGION);
       PUT FILE(SYSPRINT) SKIP EDIT (BOOK) (A);
       GO TO NEXT;

 FINISH: CLOSE FILE(STOCK);
END CRR2;
/*
//GO.SYSLMOD DD DSN=&&GOSET,DISP=(OLD,DELETE)
//GO.LOANS DD DSN=LOANS,UNIT=SYSDA,SPACE=(12,1000),DISP=(NEW,KEEP),
//            DCB=(RECFM=F,BLKSIZE=12,DSORG=DA,KEYLEN=7),
//GO.STOCK DD DSN=STOCK,UNIT=SYSDA,SPACE=(77,1050),DISP=(NEW,KEEP),
//            DCB=(RECFM=F,BLKSIZE=77,DSORG=DA,KEYLEN=4),
//GO.SYSIN DD *
'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 85. Creating a REGIONAL(2) Data Set

```
//EX12   JOB
//STEP2  EXEC PLIXCLG,PARM.PLI='NOP',PARM,LKED='LIST'
//PLI.SYSIN  DD  *
*PROCESS MAR(1,72);
DUR2: PROC OPTIONS(MAIN);
 /*  UPDATING A REGIONAL(2) DATA SET DIRECTLY - LIBRARY LOANS*/

   DCL LOANS FILE RECORD UPDATE DIRECT KEYED ENV(REGIONAL(2));
   DCL 1   RECORD,
         2   (ISSUE,REMINDER) CHAR(6);
   DCL SYSIN FILE RECORD INPUT SEQUENTIAL;
   DCL SYSIN_REC BIT(1) INIT('1'B) STATIC;
   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),   /* YYMMDD */
         2   CARD_4 CHAR(53);
   DCL REGION CHAR(8) INIT(' ');

   ON ENDFILE(SYSIN) SYSIN_REC = '0'B;
   OPEN FILE(SYSIN), FILE(LOANS);
   READ FILE(SYSIN) INTO(CARD);

   DO WHILE(SYSIN_REC);
      SUBSTR(REGION,6) = CARD.READER;
      ISSUE,REMINDER = CARD.DATE;
      PUT FILE(SYSPRINT) SKIP EDIT (CARD) (A);
      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;
      READ FILE(SYSIN) INTO(CARD);
   END;

   CLOSE FILE(SYSIN),FILE(LOANS);
END DUR2;
/*
//GO.SYSLMOD DD DSN=&&GOSET,DISP=(OLD,DELETE)
//GO.LOANS DD DSN=LOANS,DISP=(OLD,KEEP),UNIT=SYSDA,VOL=SER=nnnnnn
//GO.SYSIN DD *
5999      003       I 781221
3083      091       I 790104
1214      049       I 790205
5999      003       A 790212
3083      091       R 790212
3517      095       X 790213
/*
```

Figure 86. Updating a REGIONAL(2) Data Set Directly

```
//EX13   JOB
//STEP3  EXEC  PLIXCLG,PARM.PLI='NOP',PARM.LKED='LIST',PARM.GO='/790308'
//PLI.SYSIN  DD  *
*PROCESS MAR(1,72);
SUR2: PROC OPTIONS(MAIN);
  /* UPDATING A REGIONAL(2) DATA SET SEQUENTIALLY - LIBRARY LOANS     */

   DCL LOANS FILE RECORD SEQUENTIAL UPDATE KEYED ENV(REGIONAL(2));
   DCL LOANS_REC BIT(1) INIT('1'B) STATIC;
   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 ENV(REGIONAL(2));
   DCL 1   BOOK,
        2   AUTHOR CHAR(25),
        2   TITLE  CHAR(50),
        2   QTY    FIXED DEC(3);
   DCL TODAY CHAR(6);  /* YY/MM/DD */
   DCL INTER FIXED DEC(5);
   DCL REGION CHAR(8);

   TODAY  = '790210';
   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);     A(3),A)
        X = X+1;

      IF REMINDER < TODAY THEN              /* ? LAST REMINDER ISSUED */
         DO;                                /*   MORE THAN A MONTH AGO*/
         INTER = (BKNO-1000)/9;             /* YES, PRINT NEW REMINDER*/
         REGION = INTER;
         READ FILE(STOCK) INTO(BOOK) KEY(BKNO||REGION);
         REMINDER = TODAY;                  /* UPDATE REMINDER DATE   */
         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 SUR2;
/*
//GO.SYSLMOD DD DSN=&&GOSET,DISP=(OLD,DELETE)
//GO.LOANS DD DSN=LOANS,DISP=(OLD,DELETE),UNIT=SYSDA,VOL=SER=nnnnnn
//GO.STOCK DD DSN=STOCK,DISP=(OLD,DELETE),UNIT=SYSDA,VOL=SER=nnnnnn
/*
```

Figure 87. Updating a REGIONAL(2) Data Set Sequentially

```
//EX14   JOB
//STEP1  EXEC  PLIXCLG,PARM.PLI='NOP',PARM.LKED='LIST'
//PLI.SYSIN  DD  *
*PROCESS MAR(1,72);
CRR3: PROC OPTIONS(MAIN);
/*   CREATING A REGIONAL(3) DATA SET - LIBRARY LOANS            */

   DCL LOANS FILE RECORD KEYED ENV(REGIONAL(3));
   DCL STOCK FILE RECORD KEYED ENV(REGIONAL(3));
   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);

    /* INITIALIZE (FORMAT) LOANS DATA SET                       */
      OPEN FILE(LOANS) DIRECT OUTPUT;
      CLOSE FILE(LOANS);
      ON ENDFILE(SYSIN) GOTO FINISH;
      OPEN FILE(STOCK) SEQUENTIAL OUTPUT;

 NEXT: GET FILE(SYSIN) SKIP LIST(NUMBER,BOOK);
    INTER = (NUMBER-1000)/2250; /* REGIONS = 0,1,2,3,4 FOR A DEVICE */
                                /* HOLDING 200 (OR MORE) BOOKS/TRACK*/
    REGION = INTER;
    WRITE FILE(STOCK) FROM(BOOK) KEYFROM(NUMBER||REGION);
    GOTO NEXT;

 FINISH: CLOSE FILE(STOCK);
END CRR3;
/*
//GO.SYSLMOD DD DSN=&&GOSET,DISP=(OLD,DELETE)
//GO.LOANS DD DSN=LOANS,UNIT=SYSDA,SPACE=(TRK,3),DISP=(NEW,KEEP),
//            DCB=(RECFM=F,BLKSIZE=12,DSORG=DA,KEYLEN=7),
//GO.STOCK DD DSN=&&STOCK,UNIT=SYSDA,SPACE=(TRK,5),DISP=(NEW,KEEP),
//            DCB=(RECFM=F,BLKSIZE=77,DSORG=DA,KEYLEN=4),
//GO.SYSIN DD *
'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 88. Creating a REGIONAL(3) Data Set

```
//EX15   JOB
//STEP2  EXEC  PLIXCLG,PARM.PLI='NOP',PARM.LKED='LIST'
//PLI.SYSIN  DD  *
*PROCESS MAR(1,72);
DUR3: PROC OPTIONS(MAIN);
 /* UPDATING A REGIONAL(3) DATA SET DIRECTLY - LIBRARY LOANS     */

   DCL LOANS FILE RECORD UPDATE DIRECT KEYED ENV(REGIONAL(3));
   DCL 1   RECORD,
         2   (ISSUE,REMINDER) CHAR(6);
   DCL SYSIN FILE RECORD INPUT SEQUENTIAL;
   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);

   ON ENDFILE(SYSIN) SYSIN_REC= '0'B;
   OPEN FILE(SYSIN),FILE(LOANS);
   READ FILE(SYSIN) INTO(CARD);

  DO WHILE(SYSIN_REC);
    ISSUE,REMINDER = DATE;
      SELECT;
         WHEN(READER < '034') REGION = '00000000';
         WHEN(READER < '067') REGION = '00000001';
         OTHERWISE            REGION = '00000002';
      END;
      SELECT(CODE);
         WHEN('I') WRITE FILE(LOANS) FROM(RECORD)
                   KEYFROM(READER||BOOK||REGION);
         WHEN('R') DELETE FILE(LOANS)
                   KEY    (READER||BOOK||REGION);
         WHEN('A') REWRITE FILE(LOANS) FROM(RECORD)
                   KEY    (READER||BOOK||REGION);
         OTHERWISE PUT FILE(SYSPRINT) SKIP LIST
                   ('INVALID CODE: ',BOOK,READER);
      END;
   PUT FILE(SYSPRINT) SKIP EDIT (CARD) (A);
   READ FILE(SYSIN) INTO(CARD);
   END;

  CLOSE FILE(SYSIN),FILE(LOANS);
END DUR3;
/*
//GO.SYSLMOD DD DSN=&&GOSET,DISP=(OLD,DELETE)
//GO.LOANS DD DSN=LOANS,DISP=(OLD,KEEP),UNIT=SYSDA,VOL=SER=nnnnnn
//GO.SYSIN DD *
5999     003       I 781221
3083     091       I 790104
1214     049       I 790205
5999     003       A 790212
3083     091       R 790212
3517     095       X 790213
/*
```

Figure 89. Updating a REGIONAL(3) Data Set Directly

```
//EX16   JOB
//STEP3  EXEC  PLIXCLG,PARM.PLI='NOP',PARM.LKED='LIST',PARM.GO='/790308'
//PLI.SYSIN  DD  *
*PROCESS MAR(1,72);
SUR3: PROC OPTIONS(MAIN);
 /* UPDATING A REGIONAL(3) DATA SET SEQUENTIALLY - LIBRARY LOANS      */

   DCL LOANS FILE RECORD SEQUENTIAL UPDATE KEYED ENV(REGIONAL(3));
   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 ENV(REGIONAL(3));
   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 = '790210';
   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);     A(3),A)
        X = X+1;

      IF REMINDER < TODAY THEN
         DO;
         INTER = (BKNO-1000)/2250;
         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 SUR3;
/*
//GO.LOANS DD DSN=LOANS,DISP=(OLD,DELETE),UNIT=SYSDA,VOL=SER=nnnnnn
//GO.STOCK DD DSN=STOCK,DISP=(OLD,DELETE),UNIT=SYSDA,VOL=SER=nnnnnn
```

Figure 90. Updating a REGIONAL(3) Data Set Sequentially

TELEPROCESSING DATA SETS

Teleprocessing in PL/I is supported by record-oriented data transmission using the Telecommunications Access Method (TCAM) and PL/I files declared with the TRANSIENT attribute. A teleprocessing data set is a queue of messages originating from or destined for remote terminals (or application programs). A PL/I TRANSIENT file allows a PL/I program to access a teleprocessing data set as an INPUT file for retrieving messages or as an OUTPUT file for writing messages.

In a teleprocessing system using TCAM, the user must write a message control program (MCP) and may write one or more message processing programs (MPPs). The MCP is part of TCAM and must be written in assembler language using macros supplied by TCAM. The MPPs are application programs and may be written in PL/I.

This section briefly describes the message control program (MCP), and the message processing program (MPP). It then describes teleprocessing organization, ENVIRONMENT options for teleprocessing, and condition handling for teleprocessing.

A TCAM overview is given in OS/VS TCAM Concepts and Applications (GC30-2049). If you want more detailed information about TCAM programming facilities, see the OS/VS TCAM Application Programmer's Guide (GC30-3036) and the OS/VS2 TCAM Programmer's Guide (GC30-2041).

MESSAGE CONTROL PROGRAM (MCP)

A TCAM message control program (MCP) controls the routing of messages originating from and destined for the remote terminals and message processing programs in your TCAM installation. Each origin or destination associated with a message is identified by a name known in the MCP, and carried within the message. The MCP routes messages to and from message processing programs and terminals by means of in-storage queues. The queues may also be on disk storage when the in-storage queue is full; this support is provided by TCAM. TCAM queues may also be simulated by sequential data sets on direct-access devices; however, the data sets cannot be accessed by your PL/I program, since PL/I supports only the use of queues.

A message may be transmitted in one of several formats, only two of which are supported by PL/I. The message format is specified in the MCP and must also be specified in your PL/I program by means of the ENVIRONMENT attribute, described later in this section.

NOTE FOR SYSTEM PROGRAMMERS: Of the several message formats allowed by a TCAM MCP, PL/I supports those represented by:

- DCBOPTCD=WUC,DCBRECFM=V for PL/I ENVIRONMENT option TP(M)
- DCBOPTCD=WC,DCBRECFM=V for PL/I ENVIRONMENT Option TP(R)

MESSAGE PROCESSING PROGRAM (MPP)

A message processing program (MPP) is an application program that retrieves messages from TCAM queues and/or writes messages to TCAM queues. An MPP allows you to provide data to a problem program from a terminal and to receive output from the program with a minimum of delay. MPPs can be written in PL/I and can perform other data processing functions in addition to teleprocessing.

An MPP for reading or writing TCAM queues is not mandatory for teleprocessing installations. If the messages you transmit do not require processing, because they are simply switched between terminals, an MPP is not required.

The following sections describe PL/I teleprocessing data sets and the PL/I language features that are used to write MPPs.

TELEPROCESSING ORGANIZATION

A teleprocessing data set comprises a queue of messages that constitute the input to a PL/I message processing program. The messages are written and retrieved sequentially; keys are used to identify the terminal or application associated with the message. The TRANSIENT attribute is required in the PL/I file declaration to specify access type. TRANSIENT indicates that the contents of the data set associated with the file are reestablished each time the data set is accessed. Records can be continually added to the data set by one program during the execution of another program that continually removes records from the data set. Thus the data set can be considered to be a continuous first-in/first-out queue through which the records pass in transit between the message control program and the message processing program.

A data set associated with a TRANSIENT file differs from one associated with a DIRECT or SEQUENTIAL file in the following ways:

- Its contents are dynamic; reading a record removes it from the data set.
- The ENDFILE condition is not defined for a TRANSIENT file. Instead, the PENDING condition is raised when the input queue is empty. This does not imply the queue will remain empty since records can be continually added.

In addition to TRANSIENT access, a teleprocessing queue may be accessed for input as a SEQUENTIAL file with consecutive organization (unless you use a READ statement option, such as EVENT, that is invalid for a TRANSIENT file). This support is provided by TCAM when it detects a request from a sequential access method (BSAM or QSAM). Your program is unaware of the fact that a TCAM queue is the source of input; you will not receive terminal identifiers in the character string referenced in the KEYTO option of the READ statement and the PENDING condition will not be raised. A teleprocessing data set can be created only by a file with TRANSIENT access.

DEFINING A TELEPROCESSING DATA SET

A teleprocessing file is defined with the attributes shown in the following declaration:

```
DCL filename FILE TRANSIENT RECORD
             INPUT | OUTPUT
             BUFFERED KEYED
             ENVIRONMENT(option-list);
```

The file attributes are described in the OS and DOS PL/I Language Reference Manual. Required attributes and defaults are shown in Figure 45 on page 123.

ENVIRONMENT OPTIONS FOR TELEPROCESSING DATA SETS

For teleprocessing applications, the ENVIRONMENT options that can be specified are TP(M|R), RECSIZE(record-length), and BUFFERS(n).

TP Option

TP specifies that the file is associated with a teleprocessing data set. A message can consist of one logical record or several logical records on the teleprocessing data set.

Syntax

```
TP(M | R)
```

TP(M)
specifies that each data transmission statement in the PL/I program transmits a complete message (which may be several logical records) to or from the data set.

TP(R)
specifies that each data transmission statement in the PL/I program transmits a single logical record, which is a segment of a complete message.

One or more PL/I data transmission statements are required to completely transmit a message. On input, the PL/I application program must determine the end of message by its own means; this may be from information embedded in the message. On output, the PL/I program must provide, for each logical record, its segment position within the message. You indicate the position by a code in the first byte of the KEYFROM value, preceding the destination ID. The valid codes and their meanings are:

1	First segment of a message
blank	Intermediate segment of a message
2	Last segment in a message
3	Only segment in a message

Selection of TP(M) or TP(R) is dependent on the message format specified in your MCP. Your system programmer can tell you which to use.

RECSIZE Option

The RECSIZE option specifies the size of the record variable (or input or output buffer, for locate mode) in the PL/I program. If the TP(M) option is used, this size should be equal to the length of all the logical records that constitute the message. If it is smaller, part of the message will be lost. If it is greater, the contents of the last part of the variable (or buffer) are undefined. If the TP(R) option is specified, this size must be the same as the logical record length.

RECSIZE must be specified.

BUFFERS Option

The BUFFERS option specifies the number of intermediate buffers required to contain the longest message to be transmitted. The buffer size is defined in the message control program. If a message is too long for the buffers specified, extra buffers must be obtained before processing can continue, which increases execution time. The extra buffers are obtained by the operating system; you need not take any action.

A TRANSIENT file can be accessed by READ, WRITE, and LOCATE statements. The EVENT option cannot be used.

The READ statement is used for input, with either the INTO option or the SET option; the KEYTO option must be given. The origin name is assigned to the variable referenced in the KEYTO option. If the origin name is shorter than the character string referenced in the KEYTO option, it is padded on the right with blanks. If the KEYTO variable is a varying-length string, its current length is set to that of the origin name. The origin name should not be longer than the KEYTO variable (if it is, it is truncated), but in any case will not be longer than 8 characters. The data part of the message or record is assigned to the variable referenced in the INTO option, or the pointer variable referenced in the SET option is set to point to the data in the READ SET buffer.

A READ statement for the file will take the next message (or the next record from the current message) from the associated queue, assign the data part to the variable referenced in the READ INTO option (or set a pointer to point to the data in a READ SET buffer), and assign the character-string origin identifier to the variable referenced in the KEYTO option. The PENDING condition is raised if the input queue is empty when a READ statement is executed.

Either the WRITE or the LOCATE statement may be used for output; either statement must have the KEYFROM option—for files declared with the TP(M) option, the first 8 characters of the value of the KEYFROM expression are used to identify the destination, which must be a recognized terminal or program identifier. For files declared with the TP(R) option, indicating multiple-segment messages, the first character of the value specified in the KEYFROM expression must contain the message segment code as discussed above; the next 8 characters of the value are used to identify the destination. The data part of the message is transmitted from the variable referenced in the FROM option of the WRITE statement; or, in the case of LOCATE, a pointer variable is set to point to the location of the data in the output buffer.

The statements and options permitted for TRANSIENT files are given in Figure 91. Some examples follow:

```
DECLARE (IN INPUT,OUT OUTPUT) FILE
   TRANSIENT ENV(TP(M) RECSIZE(124)),
   (INREC, OUTREC) CHARACTER(120)
   VARYING, TERM CHARACTER(8);

READ FILE(IN) INTO(INREC) KEYTO(TERM);
   .
   .
   .
WRITE FILE(OUT) FROM(OUTREC)
  KEYFROM(TERM);
```

The above example illustrates the use of move mode in teleprocessing applications. The files IN and OUT are given the attributes KEYED and BUFFERED because TRANSIENT implies these attributes. The TP(M) option indicates that a complete message will be transmitted. The input buffer for file IN contains the next message from the input queue.

The READ statement moves the message or record from the input buffer into the variable INREC. The character string identifying the origin is assigned to TERM. If the buffer is empty when the READ statement is executed (that is, if there are no messages in the queue), the PENDING condition is raised. The implicit action for the condition is described under "Condition Handling" on page 219.

File declaration[1]	Valid statements, with options that must appear	Other options that can also be used
TRANSIENT INPUT	READ FILE(file-reference) INTO(reference) KEYTO(reference); READ FILE(file-reference) SET(pointer-reference) KEYTO(reference);	
TRANSIENT OUTPUT	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression) LOCATE based-variable FILE(file-reference) KEYFROM(expression);	 SET(pointer-reference)

Note to Figure 91:

[1]The complete file declaration would include the attributes FILE, RECORD, KEYED, BUFFERED, and the ENVIRONMENT attribute with either the TP(M) or the TP(R) option.

Figure 91. Statements and Options Permitted for TRANSIENT Files

After processing, the message or record is held in OUTREC. The WRITE statement moves it to the output buffer, together with the value of TERM (which still contains the origin name unless another name has been assigned to it during processing). From the buffer, the message is transmitted to the correct queue for the destination, as specified by the value of TERM.

The next example is similar to the previous one, except that locate mode input is used.

```
DECLARE (IN INPUT,OUT OUTPUT) FILE
   TRANSIENT ENV(TP(M) RECSIZE(124)),
   MESSAGE CHARACTER(120) VARYING
   BASED(INPTR),
   TERM CHARACTER(8);

READ FILE(IN) SET(INPTR) KEYTO(TERM);
   .
   .
   .
WRITE FILE(OUT) FROM(MESSAGE)
  KEYFROM(TERM);
```

The message data is processed in the input buffer, using the based variable MESSAGE, which has been declared with the pointer reference INPTR. (The variable MESSAGE will be aligned on a doubleword boundary.) The WRITE statement moves the processed data from the input to the output buffer; otherwise its effect is as described for the WRITE statement in the first example.

The technique used in this example would be useful in applications where the differences between processed and unprocessed messages were relatively simple, since the maximum size of input and output messages would be the same. If the length and structure of the output message could vary widely, depending on the text of the input message, locate mode output

could be used to advantage; after the input message had been read in, a suitable based variable could be located in the output buffer (using the LOCATE statement), where further processing would take place. The message would be transmitted immediately before execution of the next WRITE or LOCATE statement for the file.

Although the EVENT option is not permitted, data transmission could be overlapped with processing in an operating system that supports multitasking by means of the PL/I multitasking facilities (described in the OS and DOS PL/I Language Reference Manual). For example, the processing program could consist of a number of subtasks, each associated with a different queue. Each subtask processes the input from its associated queue, and transmits output to the required destination.

CONDITION HANDLING

The conditions that can be raised during teleprocessing transmission are TRANSMIT, KEY, RECORD, ERROR, and PENDING.

The TRANSMIT condition can be raised on input or output, as described for other types of transmission. In addition, for a TRANSIENT OUTPUT file, TRANSMIT can be raised in the following circumstances:

- The destination queue is full; TCAM rejected the message.
- For a file declared with the TP(R) ENVIRONMENT option, message segments were presented out of sequence.

The RECORD condition is raised in the same circumstances as for other types of transmission. The messages and records are treated as V-format records.

The ERROR condition is raised as for other types of transmission; it is also raised when the expression in the KEYFROM option is missing or detectably invalid.

The KEY condition is raised if the expression in the KEYFROM option is syntactically valid but does not represent an origin or a destination name recognized by the MCP.

The PENDING condition is raised only during execution of a READ statement for a TRANSIENT file. When the PENDING condition is raised, the value returned by the ONKEY built-in function is a null string. The PL/I implicit action for the PENDING condition is as follows:

- If there is no ON-unit for the PENDING condition, the PL/I transmitter module waits for a message.
- If there is an ON-unit for the PENDING condition, and it executes a normal return, the transmitter waits for a message.
- If there is an ON-unit for the PENDING condition, and it does not return normally, the next execution of a READ statement again raises PENDING if no records have been added to the queue.

There is no PL/I condition associated with the occurrence of the last segment of a message. When the TP(R) option is specified, indicating multiple-segment messages, the user is responsible for arranging the recognition of the end of the message.

ESSENTIAL INFORMATION

To access a teleprocessing data set, the file name or value of the TITLE option on the OPEN statement must be the name of a DD statement that identifies the message queue in the QNAME parameter. For example:

```
//PLIFILE  DD  QNAME=process name
```

Process name is the symbolic name of the TPROCESS macro, coded in your MCP, that defines the destination queue through which your messages will be routed. Your system programmer can provide the queue names to be used for your application.

For TRANSIENT OUTPUT files, the element expression specified in the KEYFROM option must have as its value a terminal or program identifier known to your MCP. When the TP(R) ENVIRONMENT option has been specified, indicating multiple-segment messages, the position of record segments within a message must be indicated, as described above.

EXAMPLE OF A PL/I MPP

An example of an MPP and the job control language required to run it is shown in Figure 91. The EXEC statement in the first part of the figure invokes the cataloged procedure PLIXCL to compile and link-edit the PL/I message processing program. The load module is stored in the library SYS1.MSGLIB under the member name MPPROC.

Part 1. Compiling and Link Editing the MPP

```
//JOBNAME    JOB
//  EXEC  PLIXCL
//PLI.SYSIN DD *
  MPPROC: PROC OPTIONS(MAIN);
          DCL INMSG FILE RECORD KEYED TRANSIENT ENV(TP(M) RECSIZE(100)),
              OUTMSG FILE RECORD KEYED TRANSIENT ENV(TP(M) RECSIZE(500)),
              INDATA CHAR(100),
              OUTDATA CHAR(500),
              TKEY CHAR(6);
                .
                .
                .
          OPEN FILE(INMSG) INPUT,FILE(OUTMSG) OUTPUT;
                .
                .
                .
          READ FILE(INMSG) KEYTO(TKEY) INTO(INDATA);
                .
                .
                .
          WRITE FILE(OUTMSG) KEYFROM(TKEY) FROM(OUTDATA);
                .
                .
                .
 ENDTP:   CLOSE FILE(MPP),FILE(OUTMSG);
          END MPPROC;
/*
//LKED.SYSLMOD DD DSNAME=SYS1.MSGLIB(MPPROC),DISP=OLD
```

Part 2. Executing the MPP

```
//JOBNAME    JOB  ...
//JOBLIB     DD   DSNAME=SYS1.MSGLIB(MPROC),DISP=SHR
//            EXEC  PGM=MPPROC
//INMSG   DD  QNAME=(INQUIRY)
//OUTMSG  DD  QNAME=(RESPONSE)
```

Figure 92. PL/I Message Processing Program

In the PL/I program, INMSG is declared as a teleprocessing file that can process messages up to 100 bytes long. Similarly, OUTMSG is declared as a teleprocessing file that can process messages up to 500 bytes long.

The READ statement gets a message from the queue. The terminal identifier, which is passed as a key by TCAM, is inserted into TKEY, the character string referenced in the KEYTO option. The record is placed in the INDATA variable for processing. The appropriate READ SET statement could also have been used here. The statements that process the data and place it in OUTDATA are omitted to simplify the example.

The WRITE statement moves the data from OUTDATA into the destination queue; the terminal identifier is taken from the character string in TKEY. An appropriate LOCATE statement could also have been used.

The MPP is executed in the second part of the example; the INMSG and OUTMSG DD statements associate the PL/I files MPP and OUTMSG with their respective main storage queues, that is, INQUIRY and RESPONSE.

CHAPTER 7. USING VSAM DATA SETS FROM PL/I

This chapter describes VSAM (the Virtual Storage Access Method) organization for record-oriented data transmission, the VSAM ENVIRONMENT options, compatibility with other PL/I data set organizations, and describes the statements used to load and access the three types of VSAM data sets—entry-sequenced, key-sequenced, and relative record. The chapter is concluded by a series of examples showing the PL/I statements, Access Method Services commands, and JCL statements necessary to create and access VSAM data sets.

Appendix A, "VSAM Background" on page 383 gives an introduction to VSAM and Access Method Services. It briefly describes the commands for defining and deleting data sets and for building alternate indexes. If you are not familiar with VSAM and its facilities, you may wish to read Appendix A, "VSAM Background" on page 383 before proceeding.

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.

VSAM ORGANIZATION

VSAM provides three types of data sets:

- Key-sequenced data sets (KSDS)
- Entry-sequenced data sets (ESDS)
- Relative record data sets (RRDS)

These correspond roughly to PL/I indexed, consecutive, and regional data set organizations, respectively. They are all ordered, and they can all have keys associated with their records. Both sequential and keyed access are therefore possible with all three types.

Although only key-sequenced data sets have keys as part of their logical records, keyed access is also possible for entry-sequenced data sets (using relative-byte addresses) and relative record data sets (using relative record numbers).

All VSAM data sets are held on direct-access storage devices, and a virtual storage operating system is required to use them.

The physical organization of VSAM data sets differs from those used by other access methods. VSAM does not use the concept of blocking, and, except for relative record data sets, records need not be of a fixed length. In data sets with VSAM organization, the data items are arranged in control intervals, which are in turn arranged in control areas. For processing purposes, the data items within a control interval are arranged in logical records. A control interval may contain one or more logical records, and a logical record may span two or more control intervals. Concern about blocking factors and record length is largely removed by VSAM although records cannot, of course, exceed the maximum specified size. VSAM allows access to the control intervals, but this type of access is not supported by PL/I.

VSAM data sets can have two types of indexes—prime and alternate. A prime index is the index to a KSDS that is established when the data set is defined; it always exists and may be the only index for a KSDS. You can have one or more alternate indexes on a KSDS or an ESDS. An alternate index on an ESDS enables it to be treated, 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 may 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.) You, however, must avoid making changes in the data set that would cause duplicate keys in the prime index or in a unique alternative index.

Before a VSAM data set is used for the first time, its structure is defined to the system by the DEFINE command of Access Method Services. The definition completely defines the type of the data set, its structure, and the space it requires. If the data set is indexed, its indexes (together with their key lengths and locations) and the index upgrade set are also defined. 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.

The three different data set types provide for three different types of data:

- Entry-sequenced data sets should be used for data that will be primarily accessed in the order in which it was created (or the reverse order).
- Key-sequenced data sets should be used when a record will normally be accessed through a key within the record (for example, a stock control file where the part number can be used to access the record).
- Relative record data sets are suitable for data in which each item has a particular number and the relevant record will normally be accessed by that number. An example might be a telephone system with a record associated with each number.

Records in all types of VSAM data sets can be accessed directly by means of a key, sequentially (either backward or forward), or in a combination of the two ways. That is, by selecting a starting point by means of a key and then reading forward or backward from that point.

Key-sequenced and entry-sequenced data sets can both have alternate indexes created for them. Thus they can be accessed in many sequences or by one of many keys. For example, a data set held or indexed in order of employee number could be indexed by name in an alternate index and could then be accessed in alphabetic order, in reverse alphabetic order, or directly using the name as a key, as well as in the same kind of combinations by employee number.

Figure 93 on page 225 shows how the same data could be held in the three different types of VSAM data sets and illustrates their respective advantages and disadvantages.

KEYS FOR VSAM DATA SETS

All VSAM data sets can have keys associated with their records. For key-sequenced data sets, and for entry-sequenced data sets accessed via an alternate index, the key is a defined field within the logical record. For entry-sequenced data sets, the key is the relative byte address (RBA) of the record. For relative-record data sets, the key is a relative record number.

Keys for Indexed VSAM Data Sets

Keys for key-sequenced data sets and for entry-sequenced data sets accessed via an alternate index are part of the logical records recorded on the data set. The length and location of the keys are defined when the data set is created.

The ways in which the keys may be referenced in the KEY, KEYFROM, and KEYTO options are as described under "KEY(expression) Option," "KEYFROM(expression) Option," and "KEYTO(reference) Option" in Chapter 12 of the OS and DOS PL/I Language Reference Manual. See also "Embedded Keys" on page 169.

Relative Byte Addresses (RBA)

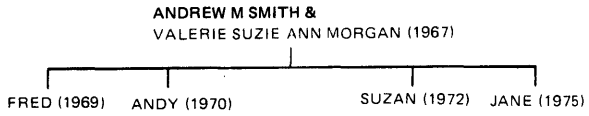
Relative byte addresses allow you to use keyed access on an ESDS associated with a KEYED SEQUENTIAL file. The RBAs, or keys, are character strings of length 4, and their values are defined by VSAM. RBAs cannot be constructed or manipulated in PL/I; their values, however, can be compared in order to determine the relative positions of records within the data set. RBAs are not normally printable.

The RBA for a record can be obtained by means of the KEYTO option, either on a WRITE statement when the data set is being loaded or extended, or on a READ statement when the data set is being read. An RBA obtained in either of these ways can subsequently be used in the KEY option of a READ or REWRITE statement.

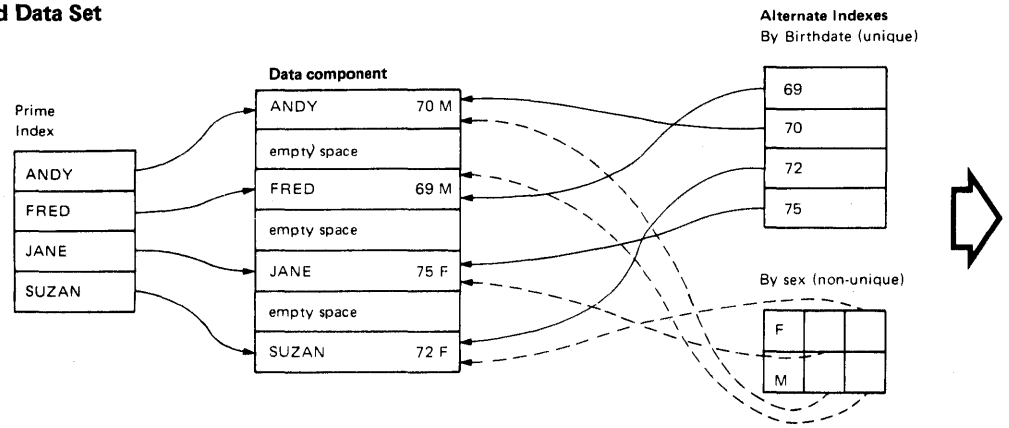
An RBA must not be used in the KEYFROM option of a WRITE statement.

VSAM allows use of the relative byte address as a key to a KSDS, but this use is not supported by PL/I.

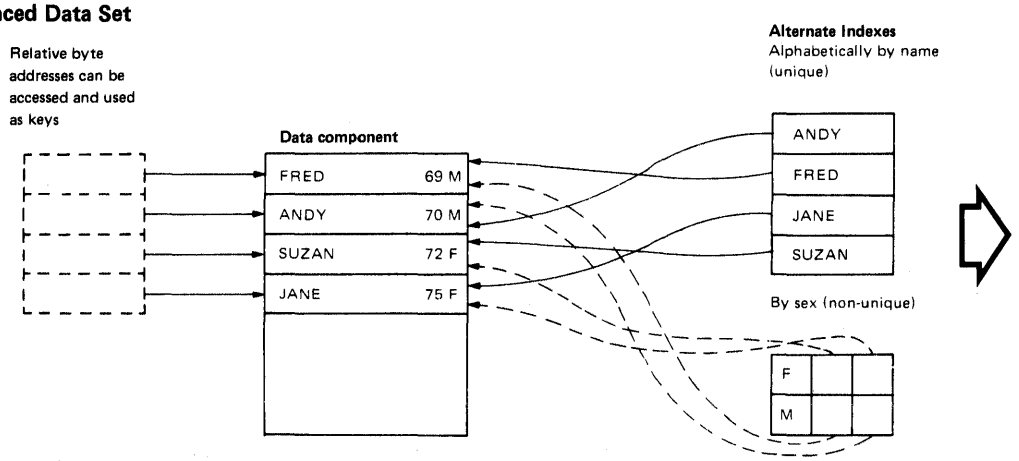
The diagrams show how the information contained in the family tree below could be held in VSAM data sets of different types.



Key-Sequenced Data Set



Entry-Sequenced Data Set



Relative Record Data Set

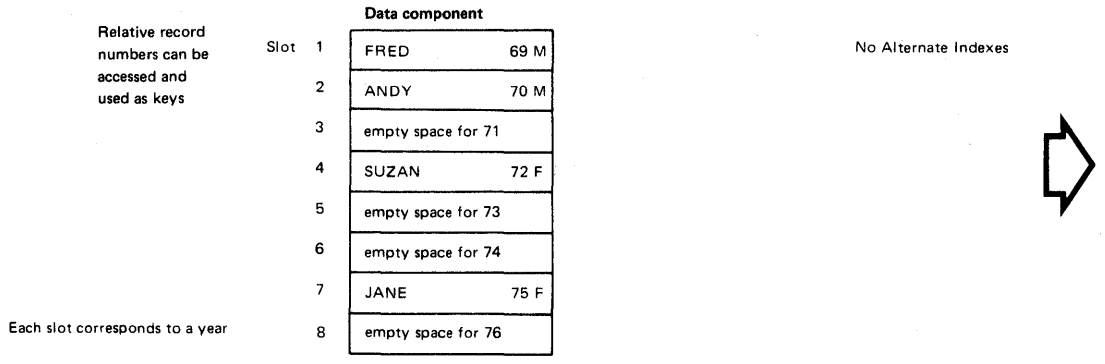


Figure 93 (Part 1 of 2). Types and Advantages of VSAM Data Sets

	Method of Loading	Method of Reading	Method of Updating	Pros and Cons
Key-Sequenced Data Set	Sequentially in order of prime index which must be unique	KEYED by specifying key of record in prime or unique alternate index SEQUENTIAL backwards or forwards in order of any index Positioning by key followed by sequential reading either backwards or forward	KEYED specifying a unique key in any index SEQUENTIAL following positioning by unique key Deletion of records allowed Insertion of records allowed	*Advantages* Complete access and updating *Disadvantages* **Records must be in order of prime index before loading** *Uses* For uses where access will be related to key
Entry-Sequenced Data Set	Sequentially (forwards only) The RBA of each record can be obtained and used as a key	SEQUENTIAL backwards or forwards KEYED using unique alternate index or RBA Positioning by key followed by sequential either backwards or forwards	New records at end only Existing records cannot have length changed Access may be sequential or KEYED using alternate index Deletion of records 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 Data Set	Sequentially starting from slot 1 KEYED specifying number of slot Positioning by key followed by sequential writes	KEYED specifying numbers as key Sequential forwards or backwards omitting empty records	Sequentially starting at a specified slot and continuing with next slot Keyed specifying numbers as key Deletion of records allowed Insertion of records 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

Figure 93 (Part 2 of 2). Types and Advantages of VSAM Data Sets

Relative Record Numbers

Records in an RRDS are identified by a relative record number that starts at 1 and is incremented by 1 for each succeeding record. These relative records numbers may be used as keys to allow 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 used 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.

CHOICE OF DATA SET TYPE

When planning your program, the first decision to be made is which type of data set to use. As discussed in Chapter 6, "Using Consecutive, Indexed, Regional, and Teleprocessing Data Sets" on page 149, there are three types of VSAM data sets and five types of non-VSAM data sets available to you. VSAM data sets can provide all the function of the other types of data sets, plus additional function available only in VSAM. VSAM can usually match other data set types in performance, and often improve upon it. However, VSAM is more subject to performance degradation through misuse of function.

The comparison of all eight types of data sets given in Figure 54 on page 149 is helpful; however, many factors in the choice of data set type for a large installation are beyond the scope of this book.

Figure 93 on page 225 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. You should follow a procedure similar to the one suggested below to help ensure a combination of data sets and indexes that provide the function you require.

1. Determine the type of data and how it will be accessed.

   a. Primarily sequentially—favors ESDS

   b. Primarily by key—favors KSDS

   c. Primarily by number—favors RRDS

2. Determine how the data set will be loaded. Note that a KSDS must be loaded in key sequence; thus an ESDS with an alternate index path may 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 do, determine whether the alternate index will have unique or nonunique keys. Use of nonunique keys may limit key processing. Conversely, the prediction that all future records will have unique keys may not be practical, and an attempt to insert a record with a nonunique key in an index that has been created for unique keys 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 94 on page 228 and Figure 95 on page 228 may be helpful.

Do not try to access a dummy VSAM data set, because you will receive an error message indicating that you have an undefined file.

Figure 96 on page 237, Figure 97 on page 240, and Figure 98 on page 244 show the statements permitted for entry-sequenced data sets, indexed data sets, and relative record data sets, respectively.

	SEQUENTIAL	KEYED SEQUENTIAL	DIRECT
INPUT	ESDS KSDS RRDS Path(N) Path(U)	ESDS KSDS RRDS Path(N) Path(U)	KSDS RRDS Path(U)
OUTPUT	ESDS RRDS	ESDS KSDS RRDS	KSDS RRDS Path(U)
UPDATE	ESDS KSDS RRDS Path(N) Path(U)	ESDS KSDS RRDS Path(N) Path(U)	KSDS RRDS Path(U)

Key: ESDS Entry-sequenced data set
KSDS Key-sequenced data set
RRDS Relative record data set
Path(N) Alternate index path with nonunique keys
(See "Alternate Index Paths" on page 391 for details.)
Path(U) Alternate index path with unique keys

The attributes on the left can be combined with those at the top of the figure for the data sets and paths shown. For example, only an ESDS and an RRDS may be SEQUENTIAL OUTPUT.

PL/I does not support dummy VSAM data sets.

Figure 94. VSAM Data Sets and Permitted File Attributes

Base Cluster Type	Alternate Index Key Type	Processing	Restrictions
KSDS	Unique key	As normal KSDS	May not modify key of access.
	Nonunique key	Limited keyed access	May not modify key of access.
ESDS	Unique key	As KSDS	No deletion. May not modify key of access.
	Nonunique key	Limited keyed access	No deletion. May not modify key of access.

Figure 95. Processing Allowed on Alternate Index Paths

DEFINING A VSAM DATA SET TO PL/I

A sequential VSAM data set is defined by a file declaration with the following attributes:

```
DCL filename FILE RECORD
             INPUT | OUTPUT | UPDATE
             SEQUENTIAL
             BUFFERED
            [KEYED]
             ENVIRONMENT(options);
```

A direct VSAM data set is defined by a file declaration with the following attributes:

```
DCL filename FILE RECORD
             INPUT | OUTPUT | UPDATE
             DIRECT
             UNBUFFERED
            [KEYED]
             ENVIRONMENT(options);
```

Figure 45 on page 123 shows the default attributes. The file attributes are described in the OS and DOS PL/I Language Reference Manual. 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 94 on page 228 shows the compatible combinations.

ENVIRONMENT OPTIONS FOR VSAM DATA SETS

Many of the options of the ENVIRONMENT attribute affecting data set structure are superfluous for VSAM data sets. If they are specified, 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:

```
VSAM
BKWD
BUFND(n)
BUFNI(n)
BUFSP(n)
PASSWORD(password-specification)
REUSE
SIS
SKIP
COBOL
GENKEY
SCALARVARYING
```

COBOL, GENKEY, and SCALARVARYING have the same effect as 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. NCP has meaning when using the ISAM compatibility interface. Figure 45 on page 123 shows which options are ignored for VSAM. Figure 45 on page 123 also shows the required and default options.

VSAM Option

Specify the VSAM option for VSAM data sets, unless the file may also access non-VSAM data sets (if this is the case, see "The VSAM Compatibility Interface" on page 234).

Syntax

```
VSAM
```

PASSWORD Option

When a VSAM data set is defined to the system (using the DEFINE command of Access Method Services), READ and UPDATE passwords can be associated with it. From that point on, the appropriate password must be included in the declaration of any PL/I file used to access the data set. The syntax of the option is:

Syntax

```
PASSWORD(password-specification)
```

password-specification
is a character constant or character variable that specifies the password for the type of access your program requires. If the specification is a constant, it must not contain a repetition factor; if it is 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. The number of chances to be allowed is specified when the data set is defined. After this number of unsuccessful tries, the UNDEFINEDFILE condition is raised.

The three levels of password supported by PL/I are:

- Master
- Update
- Read

These three levels are defined in Appendix A, "VSAM Background" on page 383. Specify the highest level of password needed for the type of access that your program will perform.

GENKEY Option

For the description of this option, see "GENKEY Option—Key Classification" on page 129.

REUSE Option

The REUSE option specifies that an OUTPUT file associated with a VSAM data set is to be used as a workfile.

Syntax

```
REUSE
```

The data set is treated as an empty data set each time the file is opened. Any secondary allocations for the data set are released, and the data set is treated exactly as if it were being opened for the first time.

A file with the REUSE option must not be associated with a data set that has alternate indexes or the BKWD option, and must not be opened for INPUT or UPDATE.

The REUSE option takes effect only if REUSE was specified in the Access Method Services DEFINE CLUSTER command.

BKWD Option

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

Syntax

```
BKWD
```

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

The BKWD option must not be specified with either the REUSE option or the GENKEY option. Also, the WRITE statement is not allowed for files declared with the BKWD option.

PERFORMANCE OPTIONS

SKIP, SIS, BUFND, BUFNI, and BUFSP are options you can specify to optimize VSAM's performance. The buffer options can also be specified in the AMP parameter of the DD statement; they are explained in your Access Method Services manual.

SKIP Option

The SKIP option of the ENVIRONMENT attribute specifies that the VSAM OPTCD "SKP" is to be used wherever possible. It is applicable to key-sequenced data sets accessed by means of a KEYED SEQUENTIAL INPUT or UPDATE file.

Syntax

```
SKIP
```

If the application program is designed to access individual records scattered throughout the data set, but the access will be primarily in ascending key order, the SKIP option should by specified for the file.

If the program is designed to read large numbers of records sequentially, without the use of the KEY option, or if it is designed to insert large numbers of records at specific points in the data set (mass sequential insert), the SKIP option should be omitted.

It is never an error to specify (or omit) the SKIP option; its effect on performance is significant only in the circumstances described.

SIS Option

The SIS option is applicable to key-sequenced data sets accessed by means of a DIRECT file.

Syntax

```
SIS
```

If mass sequential insert is used for a VSAM data set, that is, if records with ascending keys are inserted, a KEYED SEQUENTIAL UPDATE file is normally appropriate. In this case, however, VSAM delays writing the records to the data set until a complete control interval has been built. If DIRECT is specified, VSAM writes each record as soon as it is presented. Thus, in order to achieve immediate writing and faster access with efficient use of disk space, a DIRECT file should be used and the SIS option should be specified.

The SIS option is intended primarily for use in online applications.

It is never an error to specify (or omit) the SIS option; its effect on performance is significant only in the circumstances described.

BUFND Option

The BUFND option specifies the number of data buffers required for a VSAM data set. The syntax of the option is:

Syntax

```
BUFND(n)
```

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 long groups of contiguous records are to be processed sequentially.

BUFNI Option

The BUFNI option specifies the number of index buffers required for a VSAM key-sequenced data set. The syntax of the option is:

Syntax

```
BUFNI(n)
```

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

The BUFSP option specifies, in bytes, the total buffer space required for a VSAM data set (for both the data and index components). The syntax of the option is:

Syntax

```
BUFSP(n)
```

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.

FILES FOR BOTH VSAM AND NON-VSAM DATA SETS

In most cases, existing PL/I programs using files declared with ENVIRONMENT (CONSECUTIVE) or ENVIRONMENT(INDEXED) or with no ENVIRONMENT are able to access VSAM data sets without alteration. Programs using REGIONAL files must be altered and recompiled before they can use VSAM data sets. PL/I can detect that a VSAM data set is being opened, and can provide the correct access, either directly or by use of a compatibility interface. This support is provided under TSO on OS/VS2 MVS. This support is not provided in the TSO environment under OS/VS2 SVS in those cases in which DD information is supplied by the ALLOCATE command.

Existing PL/I programs that use REGIONAL(1) files cannot be used unaltered to access VSAM relative-record data sets.

The aspects of compatibility that affect the VSAM user who has data sets or programs created for other access methods are as follows:

- The re-creation of existing data sets as VSAM data sets. The Access Method Services REPRO command re-creates data sets in VSAM format. This command is described in the Access Method Services manual.
- All VSAM key-sequenced data sets have embedded keys, even if they have been converted from ISAM data sets with nonembedded keys.
- JCL DD statement changes.
- The use of programs written for non-VSAM data sets with VSAM data sets without alteration of the programs. This is described in the following section.
- The alteration of existing programs 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.

It should be realized, however, that there is no concept of fixed-length records in VSAM. Therefore, if the 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 declared 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 may not always give the required result. To ensure complete compatibility with PL/I ENV(INDEXED) files, VSAM provides the compatibility interface—a program that simulates ISAM-type handling of VSAM data sets.

Because VSAM does not support EXCLUSIVE files, programs that rely on this feature will not be compatible on VSAM and ISAM.

THE VSAM COMPATIBILITY INTERFACE

The compatibility interface simulates ISAM-type handling on VSAM key-sequenced data sets. This allows compatibility for any program whose logic depends on ISAM-type record handling.

The compatibility interface is used when the RECFM or OPTCD keyword is specified in a DD statement associated with a file declared with the INDEXED ENVIRONMENT option, or when an NCP value greater than 1 is used in the ENVIRONMENT option. These conditions are taken by the PL/I library routines to mean that the compatibility interface is required. The RECFM value, either F, V, or VS, should be chosen to match the type of record that would be used by an ISAM data set. The OPTCD value "OPTCD=I," which is the default, should be used if complete ISAM compatibility is required (see 3).

The compatibility interface cannot be used for a data set having a nonzero RKP (KEYLOC) and RECFM=F. Programs using such files must be recompiled to change the INDEXED file declaration to VSAM.

The compatibility interface is needed in the following circumstances:

1. If your program uses nonembedded keys.

2. If your program relies on the raising of the RECORD condition when an incorrect-length record is encountered.

3. If your program relies on checking for deleted records. In ISAM, deleted records remain in the data set but are flagged as deleted. In VSAM, they become inaccessible to you, and their space is available for overwriting.

   Note on Deletion: If you want the compatibility interface but want deletion of records handled in the VSAM manner, you must use OPTCD='IL' in the DD statement.

An example of DD statements that would result in the compatibility interface being used when accessing a VSAM data set is:

```
//PLIFILE DD DSNAME=VSAM1,
//           DISP=OLD,AMP='RECFM=F'
```

or, to use the compatibility interface with VSAM-type deletion of records:

```
//PLIFILE DD DSNAME=VSAM1,
//           DISP=OLD,AMP='OPTCD=IL'
```

ADAPTING EXISTING PROGRAMS FOR VSAM DATA SETS

Existing programs with indexed, consecutive, or REGIONAL(1) files can readily be adapted for use with VSAM data sets. As indicated above, programs with consecutive files may need no alteration, and there is never any necessity to alter programs with indexed files unless you wish to avoid the use of the compatibility interface or if the logic depends on EXCLUSIVE files. Programs with REGIONAL(1) data sets require only minor revision. Programs with REGIONAL(2) or REGIONAL(3) files will need restructuring before they can be used with VSAM data sets.

CONSECUTIVE Files

If the logic of the program depends on the raising of 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

Programs using indexed (that is, ISAM) files need only be changed if you wish to avoid using the compatibility interface.

Dependence on the RECORD condition should be removed, and your own code inserted to check for record length if this is necessary.

Any checking for deleted records should be removed.

REGIONAL(1) Files

Programs using REGIONAL(1) data sets can be altered to use VSAM relative record data sets.

REGIONAL(1) and any other non-VSAM ENVIRONMENT options should be removed from the file declaration and be replaced by ENV(VSAM).

Any checking for deleted records should be removed, because VSAM deleted records are not accessible to you.

ASSOCIATING SEVERAL VSAM FILES WITH ONE DATA SET

Multiple files are associated with one VSAM data set in the following ways:

- The files are associated with a common DD statement. The TITLE option of the OPEN statement can be used for this purpose, as described in "Associating Data Sets With Files" on page 119.
- The files are associated with separate DD statements, the DD statements reference the same data set name, and MACRF=DSN was specified in the VSAM OPEN request. PL/I opens all VSAM data sets with MACRF=DSN, which specifies VSAM is to 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 Programmer's Guide and normally need not concern you.

Multiple files may perform retrievals against a single data set with no difficulty. However, if one or more files perform updates, the following may occur:

- There is a risk that other files will retrieve down-level records. This can be avoided 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 the 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 attempts to access one of the unavailable records. Your application could be designed to retry the retrieval after completion of the other file's data transmission, or the error can be avoided 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 may update the alternate index before the data record is written, resulting in a mismatch between the index and the data.

SHARED DATA SETS

PL/I does not support cross-region or cross-system sharing of data sets. These types of sharing are discussed in Appendix A, "VSAM Background" on page 383 and further described in your Access Method Services manual.

HOW TO EXECUTE A PROGRAM USING VSAM DATA SETS

Before you execute a program that accesses a VSAM data set, you need to know:

- The name of the VSAM data set.
- The name of the PL/I file.
- Whether you intend to share the data set with other users (see the discussion of "Sharing a Data Set between Jobs" on page 390).

You can then write the required DD statement to access the data set:

```
//filename DD DSNAME=dsname,DISP=OLD|SHR
```

For example, if your file was called PL1FILE, your data set VSAMDS, and you wanted exclusive control of the data set, you would enter:

```
//PL1FILE DD DSNAME=VSAMDS,DISP=OLD
```

If you wanted to share your data set, you would use DISP=SHR.

If you are using a PL/I program that was originally written for ISAM data sets and requires a simulation of ISAM data set handling, you need to use the AMP parameter of the DD statement. You may also wish to use it to optimize VSAM's performance.

If you wish to optimize VSAM's performance by controlling the number of VSAM buffers used for your data set, read the section "Optimizing VSAM's Performance" in the OS/VS Virtual Storage Access Method (VSAM) Programmer's Guide.

ASSOCIATING AN ALTERNATE INDEX PATH WITH A FILE

When using an alternate index, you simply specify the name of the path in the DSNAME parameter of the DD statement associating the base data set/alternate index pair with your PL/I file. Before using an alternate index, you should be aware of the restrictions on processing; these are summarized in Figure 95 on page 228. The method used for defining a path and building an alternate index is given in "Alternate Index Paths" on page 391.

Assuming that a PL/I file was called PL1FILE and the alternate index path was called PERSALPH, the DD statement required would be:

```
//PL1FILE DD  DSNAME=PERSALPH,DISP=OLD
```

ENTRY-SEQUENCED DATA SETS

The statements and options allowed for files associated with an ESDS are shown in Figure 96.

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.

The KEYTO option may be used to obtain the relative byte address of each record as it is written. The keys thus obtained may subsequently be used to achieve keyed access to the data set.

Sequential Access

A SEQUENTIAL file that is used to access an ESDS may be opened with either the INPUT or the UPDATE attribute. If either of the options KEY or KEYTO is used, the file must also have the KEYED attribute.

Sequential access is in the order in which the records were originally loaded into the data set. The KEYTO option may be used on the READ statements to recover the RBAs of the records that are read. If the KEY option is used, the record that is recovered is the one with the specified RBA. 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 the KEY option is used; otherwise, it is the record accessed on the previous READ. A REWRITE statement must not attempt to change the length of the record that is being replaced.

The DELETE statement is not allowed for entry-sequenced data sets.

File declaration[1]	Valid statements, with options that must appear	Other options that can also be used
SEQUENTIAL OUTPUT BUFFERED	WRITE FILE(file-reference) FROM(reference);	KEYTO(reference)
	LOCATE based-variable FILE(file-reference);	SET(pointer-reference)

Figure 96 (Part 1 of 2). Statements and Options Permitted for Loading and Accessing VSAM Entry-sequenced Data Sets

File declaration[1]	Valid statements, with options that must appear	Other options that can also be used
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] or 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]
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]

Figure 96 (Part 2 of 2). Statements and Options Permitted for Loading and Accessing VSAM Entry-sequenced Data Sets

Notes to Figure 96:

1 The complete file declaration would include the attributes FILE, RECORD, and ENVIRONMENT; if either of the options KEY or KEYTO is used, 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.

KEY-SEQUENCED AND INDEXED ENTRY-SEQUENCED DATA SETS

The statements and options permitted for indexed VSAM data sets are shown in Figure 97 on page 240. An indexed data set may 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.

Loading a KSDS

When a KSDS is being loaded, the associated file must be opened for KEYED SEQUENTIAL OUTPUT. The records must be presented in ascending key order, and the KEYFROM option must be used. Note that the prime index must be used for loading the data set; no VSAM data set can be loaded via an alternate index.

If a KSDS already contains some records, and the associated file is opened with the SEQUENTIAL and OUTPUT attributes, records may be added only at the end of the data set. The rules given in the previous paragraph apply; in particular, the first record presented must have a key greater than the highest key present on the data set.

Sequential Access

A SEQUENTIAL file that is used to access a KSDS may be opened 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). The key of a record recovered in this way can be obtained by means of the KEYTO option.

If the KEY option is used, 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. Insertions can be made anywhere in the data set, irrespective of the position of any previous access. If the data set is being accessed 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 in which they were added to the data set.

REWRITE statements with or without the KEY option are allowed for UPDATE files. If the KEY option is used, 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 a record is rewritten using an alternate index, the prime key of the record must not be changed.

Direct Access

A DIRECT file that is used to access an indexed VSAM data set may be opened with the INPUT, OUTPUT, or UPDATE attribute. A DIRECT file must not be used to access the data set via a nonunique index.

If a DIRECT OUTPUT file is used 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 a DIRECT INPUT or DIRECT UPDATE file is used, records may be read, written, rewritten, or deleted in the same way as for a KEYED SEQUENTIAL file.

SAMEKEY Built-In Function

If a VSAM data set is being accessed via 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.

File declaration[1]	Valid statements, with options that must appear	Other options that can also be used
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(file-reference)[5]	KEY(expression)

Figure 97 (Part 1 of 3). Statements and Options Permitted for Loading and Accessing VSAM Indexed Data sets

File declaration[1]	Valid statements, with options that must appear	Other options that can also be used
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);	

Figure 97 (Part 2 of 3). Statements and Options Permitted for Loading and Accessing VSAM Indexed Data sets

File declaration[1]	Valid statements, with options that must appear	Other options that can also be used
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)

Figure 97 (Part 3 of 3). Statements and Options Permitted for Loading and Accessing VSAM Indexed Data sets

Notes to Figure 97:

1 The complete file declaration would include the attributes FILE and RECORD. If any of the options KEY, KEYFROM, or KEYTO is used, the declaration must also include the attribute KEYED.

The EXCLUSIVE attribute for DIRECT INPUT or UPDATE files, the UNLOCK statement for DIRECT UPDATE files, or the NOLOCK option of the READ statement for DIRECT INPUT files are ignored if they are used for files associated with a VSAM KSDS.

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

3 A SEQUENTIAL OUPUT file must not be associated with a data set accessed via an alternate index.

4 A DIRECT file must not be associated 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.

RELATIVE RECORD DATA SETS

The statements and options permitted for VSAM relative record data sets (RRDS) are shown in Figure 98 on page 244.

Loading an RRDS

When an RRDS is being loaded, the associated file must be opened for OUTPUT. Either a DIRECT or a SEQUENTIAL file may be used.

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 224).

For a SEQUENTIAL OUTPUT file, WRITE statements with or without the KEYFROM option may be used. If the KEYFROM option is specified, the record is placed in the specified slot; if it is omitted, the record is placed in the slot following the current position. There is no requirement for the records to be presented in ascending relative record number order. If the KEYFROM option is omitted, the relative record number of the written record can be obtained by means of the KEYTO option.

If an RRDS is to be loaded sequentially, without use of the KEYFROM or KEYTO options, the 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 the KEYFROM option is used, the KEY condition is raised; if it is omitted, the ERROR condition is raised.

Sequential Access

A SEQUENTIAL file that is used to access an RRDS may be opened with either the INPUT or the UPDATE attribute. If any of the options KEY, KEYTO, or KEYFROM is used, the file must also have the KEYED attribute.

For READ statements without the KEY option, the records are recovered in ascending relative record number order. Any empty slots in the data set are skipped.

If the KEY option is used, the record recovered by a READ statement is the one with the specified relative record number. 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. Insertions can be made anywhere in the data set, irrespective 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 the KEYFROM option is omitted, 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.

The KEYTO option may be used 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 the KEY option is used, the record that is rewritten is the record with the specified relative record number; otherwise, it is the record that was accessed by the previous READ statement.

DELETE statements, with or without the KEY option, may be used to delete records from the data set.

Direct Access

A DIRECT file used to access an RRDS may have the OUTPUT, INPUT, or UPDATE attribute. Records may be read, written, rewritten, or deleted exactly as though a KEYED SEQUENTIAL file were used.

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

Figure 98 (Part 1 of 3). Statements and Options Permitted for Loading and Accessing VSAM Relative-Record Data Sets

File declaration[1]	Valid statements, with options that must appear	Other options that can also be used
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);	

Figure 98 (Part 2 of 3). Statements and Options Permitted for Loading and Accessing VSAM Relative-Record Data Sets

File declaration[1]	Valid statements, with options that must appear	Other options that can also be used
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)

Figure 98 (Part 3 of 3). Statements and Options Permitted for Loading and Accessing VSAM Relative-Record Data Sets

Notes to Figure 98:

1 The complete file declaration would include the attributes FILE and RECORD. If any of the options KEY, KEYFROM, or KEYTO is used, the declaration must also include the attribute KEYED.

The EXCLUSIVE attribute for DIRECT INPUT or UPDATE files, the UNLOCK statement for DIRECT UPDATE files, or the NOLOCK option of the READ statement for DIRECT INPUT files are ignored if they are used for files associated with a VSAM KSDS.

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

EXAMPLES

EXAMPLES WITH ENTRY-SEQUENCED DATA SETS

The examples in Figure 99 on page 247 through Figure 103 on page 251 for ESDS are based on the family tree shown in Figure 93 on page 225.

Defining and Loading an Entry-Sequenced Data Set

In Figure 99 on page 247, the data set is defined with the DEFINE CLUSTER command and given the name PLIVSAM.AJC1. The NONINDEXED keyword causes an ESDS to be defined.

```
//OPT9#7   JOB
//STEP1    EXEC   PGM=IDCAMS,REGION=512K
//SYSPRINT DD SYSOUT=A
//SYSIN    DD    *
     DEFINE CLUSTER -
      (NAME(PL1VSAM.AJC1.BASE) -
      VOLUMES(nnnnnn) -
      NONINDEXED -
      RECORDSIZE(80 80) -
      TRACKS(2 2)) -
      CATALOG(catalog.name)
/*
//STEP2   EXEC   PLIXCLG
//PLI.SYSIN  DD  *
   CREATE: PROC OPTIONS (MAIN);

      DCL
        FAMFILE FILE SEQUENTIAL OUTPUT ENV(VSAM),
        IN FILE RECORD INPUT,
        STRING CHAR(80);

      ON ENDFILE(IN)  GOTO FINITO;

      DO I=1 BY 1;
        READ FILE(IN) INTO (STRING);
        PUT FILE(SYSPRINT) SKIP EDIT (STRING) (A);
        WRITE FILE(FAMFILE) FROM (STRING);
      END;

   FINITO:
      PUT SKIP EDIT(I-1,' RECORDS PROCESSED')(A);
    END CREATE;
//LKED.SYSLMOD    DD   DSN=HPU8.MYDS(PGMA),DISP=(NEW,CATLG),
//         UNIT=SYSDA,SPACE=(CYL,(1,1,1))
//GO.SYSLMOD DD DUMMY
//GO.FAMFILE DD   DSNAME=PL1VSAM.AJC1.BASE,DISP=OLD
//GO.IN      DD    *
FRED                    69          M
ANDY                    70          M
SUZAN                   72          F
//
```

Figure 99. Defining and Loading an Entry-Sequenced Data Set (ESDS)

The PL/I program writes the data set using a SEQUENTIAL OUTPUT file and a WRITE FROM statement. The DD statement for the file contains the DSNAME of the data set given in the NAME parameter of the DEFINE CLUSTER command.

The RBA of the records could have been obtained during the writing for subsequent use as keys in a KEYED file. To do this, a suitable variable would have to be declared to hold the key and the WRITE...KEYTO statement used. For example:

```
DCL CHARS CHAR(4);
WRITE FILE(FAMFILE) FROM (STRING)
 KEYTO(CHARS);
```

Note that the keys would not normally be printable, but could be retained for subsequent use.

The cataloged procedure PLIXCLG is used. Because the same program can be used for adding records to the data set, it is retained in a library. Its use is shown in the next example.

Updating an Entry-Sequenced Data Set

Figure 100 shows the addition of a new record on the end of an ESDS. This is done by reexecuting the program shown in Figure 99 on page 247. A SEQUENTIAL OUTPUT file is used and the data set associated with it by use of the DSNAME parameter specifying the name PL1VSAM.AJC1.BASE specified in the DEFINE command shown in Figure 99 on page 247.

```
//OPT9#8    JOB
//STEP1     EXEC  PGM=PGMA
//STEPLIB    DD   DSN=HPU8.MYDS(PGMA),DISP=(OLD,KEEP),UNIT=SYSDA,
//         VOL=SER=nnnnnn
//SYSPRINT   DD   SYSOUT=A
//FAMFILE    DD   DSN=PL1VSAM.AJC1.BASE,DISP=SHR
//IN         DD   *
JANE                      75          F
//
```

Figure 100. Updating an ESDS

Existing records can be rewritten in an ESDS, provided that the length of the record is not changed. A SEQUENTIAL or a KEYED SEQUENTIAL update file can be used to do this. If keys are used, they can be the RBAs or keys of an alternate index path.

Delete is not allowed for ESDS.

Creating a Unique Alternate Index Path for an ESDS

```
//OPT9#9   JOB
//STEP1    EXEC    PGM=IDCAMS,REGION=512K
//SYSPRINT   DD   SYSOUT=A
//SYSIN    DD    *
         DEFINE ALTERNATEINDEX -
           (NAME(PL1VSAM.AJC1.ALPHIND) -
            VOLUMES(nnnnnn) -
            TRACKS(4 1) -
            KEYS(15 0) -
            RECORDSIZE(20 40) -
            UNIQUEKEY -
           RELATE(PL1VSAM.AJC1.BASE)) -
          CATALOG(catalog.name)
/*
//STEP2    EXEC    PGM=IDCAMS,REGION=512K
//DD1      DD    DSNAME=PL1VSAM.AJC1.BASE,DISP=SHR
//DD2      DD    DSNAME=PL1VSAM.AJC1.ALPHIND,DISP=SHR
//IDCUT1   DD    AMP='AMORG',DISP=SHR,VOL=SER=nnnnnn,UNIT=SYSDA
//IDCUT2   DD    AMP='AMORG',DISP=SHR,VOL=SER=nnnnnn,UNIT=SYSDA
//SYSPRINT   DD   SYSOUT=A
//SYSIN    DD    *
         BLDINDEX  INFILE(DD1)  OUTFILE(DD2) -
          CATALOG(catalog.name)
         DEFINE PATH -
           (NAME(PL1VSAM.AJC1.ALPHPATH) -
           PATHENTRY(PL1VSAM.AJC1.ALPHIND))-
          CATALOG(catalog.name)
//
```

Figure 101. Creating a Unique Key Alternate Index Path for an ESDS

Figure 101 shows the creation of a unique key alternate index path for the ESDS defined and loaded in Figure 99 on page 247. Using this path, the data set is indexed by the name of the child in the first 15 bytes of the record. Three Access Method Services commands are used. These are:

DEFINE ALTERNATEINDEX
defines the alternate index as a data set to VSAM.

BLDINDEX
places the pointers to the relevant records in the alternate index.

DEFINE PATH
defines an entity that can be associated with a PL/I file in a DD statement.

DD statements are required for the INFILE and OUTFILE operands of BLDINDEX and for the sort files. Care should be taken that the correct names are specified at the various points. A fuller description of defining an alternate index is given in "Alternate Index Paths" on page 391.

Creating a Nonunique Key Alternate Index Path for an ESDS

Figure 102 on page 250 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.

```
//OPT9#10 JOB
//STEP1    EXEC   PGM=IDCAMS,REGION=512K
//SYSPRINT  DD   SYSOUT=A
//SYSIN     DD     *
     /*care must be taken with record size */
     DEFINE ALTERNATEINDEX -
       (NAME(PL1VSAM.AJC1.SEXIND) -
        VOLUMES(nnnnnn) -
        TRACKS(4 1) -
        KEYS(1 37) -
        NONUNIQUEKEY -
        RELATE(PL1VSAM.AJC1.BASE)) -
        RECORDSIZE(20 400)) -
      CATALOG(HB0009.VSAMCAT)
/*
//STEP2    EXEC   PGM=IDCAMS,REGION=512K
//DD1      DD  DSNAME=PL1VSAM.AJC1.BASE,DISP=OLD
//DD2      DD  DSNAME=PL1VSAM.AJC1.SEXIND,DISP=OLD
//IDCUT1   DD  AMP='AMORG',DISP=SHR,VOL=SER=nnnnnn,UNIT=SYSDA
//IDCUT2   DD  AMP='AMORG',DISP=SHR,VOL=SER=nnnnnn,UNIT=SYSDA
//SYSPRINT  DD   SYSOUT=A
//SYSIN    DD     *
    BLDINDEX  INFILE(DD1)  OUTFILE(DD2) -
      CATALOG(catalog.name)
    DEFINE PATH -
      (NAME(PL1VSAM.AJC1.SEXPATH) -
       PATHENTRY(PL1VSAM.AJC1.SEXIND))-
     CATALOG(catalog.name)
//
```

Figure 102. Creating a Nonunique Key Alternate Index Path for an ESDS

The three commands and the DD statement are as described in Figure 101 on page 249. The fact that the index has nonunique keys is specified by the use of the NONUNIQUEKEY operand. When creating an index with nonunique keys, care should be taken to ensure that the RECORDSIZE specified will be 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 may have the same key, a large record will be required.

Using Alternate Indexes and Backward Reading on an ESDS

Figure 103 on page 251 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.

SEXFLE is the alternate index path that corresponds to the sex of the children.

```
//PLI.SYSIN  DD   *
  READIT: PROC OPTIONS(MAIN);
    DCL   BASEFLE FILE SEQUENTIAL INPUT ENV(VSAM),
           /*file to read base data set forward */
          BACKFLE FILE SEQUENTIAL INPUT ENV(VSAM BKWD),
           /*file to read base data set backward */
          ALPHFLE FILE DIRECT INPUT ENV(VSAM),
           /*file to access via unique alternate index path */
          SEXFILE FILE KEYED SEQUENTIAL INPUT ENV(VSAM),
           /*file to access via nonunique alternate index path */
          STRING CHAR(80),  /*string to be read into */
          1 STRUC DEF (STRING),
            2 NAME CHAR(25),
            2 DATE_OF_BIRTH CHAR(2),
            2 FILL CHAR(10),
            2 SEX CHAR(1);
            DCL NAMEHOLD CHAR(25),SAMEKEY BUILTIN;

    /*print out the family eldest first*/
    ON ENDFILE(BASEFLE) GOTO YPRINT;
    PUT EDIT('FAMILY ELDEST FIRST')(A);

    DO WHILE('1'B);
       READ FILE(BASEFLE) INTO (STRING);
       PUT SKIP EDIT(STRING)(A);
    END;
  YPRINT:
    CLOSE FILE(BASEFLE);
    PUT SKIP(2);

     /*close before using data set from other file not
       necessary but good practice to prevent potential
       problems*/
    ON ENDFILE(BACKFLE) GOTO AGEQUERY;
    PUT SKIP(3) EDIT('FAMILY YOUNGEST FIRST')(A);

    DO WHILE('1'B);
      READ FILE(BACKFLE) INTO (STRING);
      PUT SKIP EDIT(STRING)(A);
    END;
  AGEQUERY:  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);
       GOTO SPRINT;
    END;

    ON ENDFILE(SYSIN) GOTO SPRINT;

    DO WHILE('1'B);
          GET SKIP EDIT(NAMEHOLD)(A(25));
          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);
    END;

 SPRINT:
        CLOSE FILE(ALPHFLE);
        PUT SKIP(1);
```

Figure 103 (Part 1 of 2). Alternate Index Paths and Backward Reading with an ESDS

```
   /*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:
      CLOSE FILE(SEXFILE);
 END READIT;
GO.BASEFLE   DD   DSN=PL1VSAM.AJC1.BASE,DISP=SHR
GO.BACKFLE   DD   DSN=PL1VSAM.AJC1.BASE,DISP=SHR
GO.ALPHFLE   DD   DSN=PL1VSAM.AJC1.ALPHPATH,DISP=SHR
GO.SEXFILE   DD   DSN=PL1VSAM.AJC1.SEXPATH,DISP=SHR
GO.SYSIN     DD    *
ANDY
/*
//STEP2    EXEC PGM=IDCAMS,REGION=512K
//SYSPRINT    DD    SYSOUT=A
//SYSIN       DD     *
     DELETE -
          PL1VSAM.AJC1.BASE -
          CATALOG(catalog.name)
//
```

Figure 103 (Part 2 of 2). Alternate Index Paths and Backward Reading with an ESDS

There are DD statements for all the files. They connect BASEFLE and BACKFLE to the base data set by specifying the name of the base data set in the DSNAME parameter, and connect ALPHFLE and SEXFLE by specifying the names of the paths given in Figure 101 on page 249 and Figure 102 on page 250.

The program uses SEQUENTIAL files to access the data and print it first in the normal order, then in the reverse order. At the label AGEQUERY, a DIRECT file is used to read the data associated with an alternate index key in the unique alternate index.

Finally, at the label SPRINT, a KEYED SEQUENTIAL file is used to print a list of the girls 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 girls 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. They can also be deleted separately, as described in "Using the Access Method Services Program" on page 389.

EXAMPLES WITH KEY-SEQUENCED DATA SETS

The examples in Figure 104 on page 254 through Figure 107 on page 258 show the use of a key-sequenced data set to hold a telephone directory. The prime index is by the name of the subscriber. In Figure 104 on page 254, the data set is defined and loaded. In Figure 105 on page 255, it is altered by means of a prime index. In Figure 107 on page 258, a unique key alternate index path is created using the numbers as the alternate key. In Figure 108 on page 259, use of the alternate index path is shown to update the base data set using the number as a key and to print out the data in order of the numbers. These examples can be compared with the "Examples of Indexed Data Sets" on page 186.

```
//OPT9#12  JOB
//    EXEC  PGM=IDCAMS,REGION=512K
//SYSPRINT  DD  SYSOUT=A
//SYSIN     DD  *
   DEFINE CLUSTER -
     (NAME(PL1VSAM.AJC2.BASE) -
     VOLUMES(nnnnnn) -
     INDEXED -
     TRACKS(3 1) -
     KEYS(20 0) -
     RECORDSIZE(23 80)) -
     CATALOG(catalog name)
/*
//    EXEC  PLIXCLG
//PLI.SYSIN DD *
 TELNOS: PROC OPTIONS(MAIN);

        DCL DIREC FILE RECORD SEQUENTIAL OUTPUT KEYED ENV(VSAM),
            CARD CHAR(80),
            NAME CHAR(20) DEF CARD POS(1),
            NUMBER CHAR(3) DEF CARD POS(21),
            OUTREC CHAR(23) DEF CARD POS(1);

        ON ENDFILE(SYSIN) GOTO FINISH;

        OPEN FILE(DIREC) OUTPUT;

 NEXTIN: GET FILE(SYSIN) EDIT(CARD)(A(80));
        WRITE FILE(DIRECT) FROM(OUTREC) KEYFROM(NAME);
        GOTO NEXTIN;
 FINISH: CLOSE FILE(DIREC);

        END TELNOS;
//GO.DIREC  DD  DSNAME=PL1VSAM.AJC2.BASE,DISP=OLD
//GO.SYSIN  DD  *
ACTION,G.            162
BAKER,R.             152
BRAMLEY,O.H.         248
CHEESEMAN,D.         141
CORY,G.              336
ELLIOTT,D.           875
FIGGINS,S.           413
HARVEY,C.D.W.        205
HASTINGS,G.M.        391
KENDALL,J.G.         294
LANCASTER,W.R.       624
MILES,R.             233
NEWMAN,M.W.          450
PITT,W.H.            515
ROLF,D.E.            114
SHEERS,C.D.          241
SUTCLIFFE,M.         472
TAYLOR,G.C.          407
WILTON,L.W.          404
WINSTONE,E.M.        307
//
```

Figure 104. Defining and Loading a Key-Sequenced Data Set (KSDS)

```
//OPT9#13  JOB
//STEP1 EXEC  PLIXCLG
//PLI.SYSIN   DD  *
 DIRUPDT: PROC OPTIONS(MAIN);

         DCL DIREC FILE RECORD KEYED ENV(VSAM),
             ONCODE BUILTIN,
             OUTREC CHAR(23),
             NUMBER CHAR(3) DEF OUTREC POS(21),
             NAME CHAR(20) DEF OUTREC,
             CODE CHAR(2);

         ON ENDFILE(SYSIN) GO TO PRINT;
         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;

 NEXT: GET FILE(SYSIN) EDIT (NAME,NUMBER,CODE)
         (COLUMN(1),A(20),A(3),A(1));
       PUT FILE(SYSPRINT) SKIP EDIT (' ',NAME,'#',NUMBER,' ',CODE)
       (A(1),A(20),A(1),A(3),A(1),A(1));
         IF CODE='A' THEN
            WRITE FILE(DIREC) FROM(OUTREC) KEYFROM(NAME);
         ELSE IF CODE='C' THEN
            REWRITE FILE(DIREC) FROM(OUTREC) KEY(NAME);
         ELSE IF CODE='D' THEN
            DELETE FILE(DIREC) KEY(NAME);
         ELSE
            PUT FILE(SYSPRINT) SKIP EDIT
             ('INVALID CODE: ',NAME) (A(15),A);
         GO TO NEXT;

 PRINT:  CLOSE FILE(DIREC);
         PUT FILE(SYSPRINT) PAGE;
         OPEN FILE(DIREC) SEQUENTIAL INPUT;
         ON ENDFILE(DIREC) GO TO FINISH;

 NEXTIN: READ FILE(DIREC) INTO(OUTREC);
         PUT FILE(SYSPRINT) SKIP EDIT(OUTREC)(A);
         GO TO NEXTIN;
 FINISH: CLOSE FILE(DIREC);
 END DIRUPDT;
/*
//GO.DIREC DD DSNAME=PL1VSAM.AJC2.BASE,DISP=OLD
//GO.SYSIN DD *
NEWMAN,M.W.          516C
GOODFELLOW,D.T.      889A
MILES,R.               D
HARVEY,C.D.W.        209A
BARTLETT,S.G.        183A
CORY,G.                D
READ,K.M.            001A
PITT,W.H.
ROLF,D.F.              D
ELLIOTT,D.           291C
HASTINGS,G.M.          D
BRAMLEY,O.H.         439C
//
```

Figure 105. Updating a KSDS

Defining and Loading a Key-Sequenced Data Set

Figure 104 on page 254 shows the DEFINE command used to define a KSDS. The data set is given the name PL1VSAM.AJC2.BASE and defined as a KSDS because of the use of the INDEXED operand. The position of the keys within the record is defined in the KEYS operand.

Within the PL/I program, a KEYED SEQUENTIAL OUTPUT file is used with a WRITE...FROM...KEYFROM statement. The data is presented in ascending key order. A KSDS must be loaded in this manner.

The file is associated with the data set by a DD statement which uses the name given in the DEFINE command as the DSNAME parameter.

Updating a Key-Sequenced Data Set

Figure 105 on page 255 shows one method by which a KSDS can be updated using the prime index.

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

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

At the label NEXT, the name, number, and code are read in and action taken according to the value of the code. A KEY on-unit is used to handle any incorrect keys. When the updating is finished (at the label PRINT), the file DIREC is closed and reopened with the attributes SEQUENTIAL INPUT. The file is then read sequentially and printed.

The file is associated with the data set by a DD statement that uses the DSNAME PL1VSAM.AJC2.BASE defined in the Access Method Services DEFINE CLUSTER command in Figure 104 on page 254.

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 may have been improved by use of the SKIP ENVIRONMENT option. For mass sequential insertion, a KEYED SEQUENTIAL UPDATE file should be used. 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 would be:

```
DCL DIREC KEYED SEQUENTIAL UPDATE
    ENV(VSAM);
WRITE FILE(DIREC) FROM(OUTREC)
 KEYFROM(NAME);
```

The PL/I input/output routines would detect that the keys were in sequence and make the correct requests to VSAM. If the keys were not in sequence, this too would be detected and no error would occur, although the performance advantage would be lost. VSAM in fact provides three methods of insertion as shown in Figure 106 on page 257.

Method	Requirements	Freespace	When Written Onto Data Set	PL/I Attributes Required
SEQ	Keys in sequence	Kept	Only when necessary	KEYED SEQUENTIAL UPDATE
SKP	Keys in sequence	Used	Only when necessary	KEYED SEQUENTIAL UPDATE ENV(VSAM SKIP)
DIR	Keys in any order	Used	After every statement	DIRECT
DIR(MACRF =SIS)	Keys in any order	Kept	After every statement	DIRECT ENV(VSAM SIS)

Figure 106. VSAM Methods of Insertion into a KSDS

SKIP means that the sequence must be followed but that records may be omitted. Absolute sequence or order need not be maintained if SEQ or SKIP is used because the PL/I routines determine which type of request to make to VSAM for each statement, first checking on the keys to see 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 further normal alterations to be made in that area without loss of performance. To preserve free space balance when immediate writing of the data set is required during mass sequential insertion, as it may be on interactive systems, the SIS ENVIRONMENT option should be used with DIRECT files.

Creating a Unique Alternate Index Path for a KSDS

Figure 107 on page 258 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 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 DD statement.

DD statements are required for the INFILE and OUTFILE of BLDINDEX and for the sort files. Care should be taken not to confuse the names involved. See the discussion in "BLDINDEX Command" on page 396.

When creating an alternate index with a unique key, you should ensure that no further records could be included with the same alternate key. In practice, a unique key alternate index would not be entirely satisfactory for a telephone directory as it would not allow two people to have the same number. Similarly, the prime key would prevent one person having two numbers. A solution would be to have an ESDS with two nonunique key alternate indexes, or to restructure the data format to allow more than one number per person and to have a nonunique key

alternate index for the numbers. See Figure 101 on page 249 for an example of creation of an alternate index with nonunique keys.

```
//OPT9#14   JOB
//STEP1     EXEC  PGM=IDCAMS,REGION=512K
//SYSPRINT  DD    SYSOUT=A
//SYSIN     DD    *
        DEFINE ALTERNATEINDEX -
          (NAME(PL1VSAM.AJC2.NUMIND) -
           VOLUMES(nnnnnn) -
           TRACKS(4 4) -
           KEYS(3 20) -
           RELATE(PL1VSAM.AJC2.BASE) -
           UNIQUEKEY -
           RECORDSIZE(24 48)) -
         CATALOG(catalog.name)
/*
//STEP2   EXEC  PGM=IDCAMS,REGION=512K
//SYSPRINT   DD     SYSOUT=A
//DD1     DD     DSN=PL1VSAM.AJC2.BASE,DISP=OLD
//DD2     DD     DSN=PL1VSAM.AJC2.NUMIND,DISP=OLD
//IDCUT1  DD     AMP='AMORG',DISP=OLD,UNIT=SYSDA,VOL=SER=nnnnnn
//IDCUT2  DD     AMP='AMORG',DISP=OLD,UNIT=SYSDA,VOL=SER=nnnnnn
//SYSIN     DD     *
        BLDINDEX  INFILE(DD1)  OUTFILE(DD2) -
        CATALOG(catalog.name)

        DEFINE PATH -
          (NAME(PL1VSAM.AJC2.NUMPATH) -
           PATHENTRY(PL1VSAM.AJC2.NUMIND))-
         CATALOG(catalog.name)
//
```

Figure 107. Creating an Alternate Index Path for a KSDS

Using a Unique Alternate Index Path with a KSDS

Figure 108 on page 259 shows the use of a path with a unique alternate index key to update a KSDS and then to access and print it in the order of the alternate index.

```
//OPT9#16   JOB
//STEP1  EXEC  PLIXCLG,REGION.GO=256K
//PLI.SYSIN  DD  *
   ALTER: PROC OPTIONS(MAIN);
        DCL NUMFLE1 FILE RECORD DIRECT OUTPUT ENV(VSAM),
            NUMFLE2 FILE RECORD SEQUENTIAL INPUT ENV(VSAM),
            IN FILE RECORD,
            STRING CHAR(80),
            NAME CHAR(20) DEF STRING,
            NUMBER CHAR(3) DEF STRING POS(21),
            DATA CHAR(23) DEF STRING;

        ON KEY (NUMFLE1) BEGIN;
          PUT SKIP EDIT('DUPLICATE NUMBER')(A);
        END;
        ON ENDFILE(IN) GOTO PRINTIT;

        DO WHILE('1'B);
           READ FILE(IN) INTO (STRING);
           PUT FILE(SYSPRINT) SKIP EDIT (STRING) (A);
           WRITE FILE(NUMFLE1) FROM (STRING) KEYFROM(NUMBER);
        END;

 PRINTIT:
        CLOSE FILE(NUMFLE1);
        ON ENDFILE(NUMFLE2) GOTO FINALE;

        DO WHILE('1'B);
          READ FILE(NUMFLE2) INTO (STRING);
          PUT SKIP EDIT(DATA)(A);
        END;

 FINALE:
        PUT SKIP(3) EDIT('****SO ENDS THE PHONE DIRECTORY****')(A);
  END ALTER;
/*
//GO.IN    DD   *
RIERA L             123
/*
//NUMFLE1    DD   DSN=PL1VSAM.AJC2.NUMPATH,DISP=OLD
//NUMFLE2    DD   DSN=PL1VSAM.AJC2.NUMPATH,DISP=OLD
//STEP2    EXEC PGM=IDCAMS,COND=EVEN
//SYSPRINT   DD  SYSOUT=A
//SYSIN   DD    *
    DELETE -
         PL1VSAM.AJC2.BASE -
         CATALOG(catalog.name)
//
```

Figure 108. Using a Unique Alternate Index Path to Access a KSDS

The alternate index path is associated with the PL/I file by a DD statement that specifies the name of the path (PL1VSAM.AJC2.NUMPATH, given in the DEFINE PATH command in Figure 107 on page 258) as the DSNAME.

In the first section of the program, a DIRECT OUTPUT file is used to insert a new record using the alternate index key. Note that any alteration made with an alternate index must not alter the prime key or the alternate index key of access of an existing record or add a duplicate key in the prime index or any unique key alternate index.

In the second section of the program (at the label PRINTIT), the data set is read in the order of the alternate index keys using a SEQUENTIAL INPUT file. It is then printed onto SYSPRINT.

EXAMPLES WITH RELATIVE RECORD DATA SETS

These examples show the defining and loading of an RRDS and its subsequent updating. The examples correspond with the REGIONAL(1) examples in "REGIONAL(1) Data Sets" on page 204. They use the same telephone directory data, but use the number as the key to the record. The record contains only the name.

Defining and Loading a Relative Record Data Set

In Figure 109 on page 261, the data set is defined with a DEFINE CLUSTER command and given the name PL1VSAM.AJC3.BASE. The fact that it is an RRDS is determined by the NUMBERED keyword. In the PL/I program, it is loaded with a DIRECT OUTPUT file and a WRITE...FROM...KEYFROM statement is used.

If the data had been in order and the keys in sequence, it would have been possible to use a SEQUENTIAL file and write into the data set from the start. The records would then have been placed in the next available slot and given the appropriate number. The number of the key for each record could have been returned using the KEYTO option.

The PL/I file is associated with the data set by the DD statement, which uses as the DSNAME the name given in the DEFINE CLUSTER command.

```
//OPT9#17 JOB
//STEP1 EXEC PGM=IDCAMS,REGION=512K
//SYSPRINT DD SYSOUT=A
//SYSIN    DD *
         DEFINE CLUSTER -
             (NAME(PL1VSAM.AJC3.BASE) -
             VOLUMES(nnnnnn) -
             NUMBERED -
             TRACKS(2 2) -
             RECORDSIZE(20 20)) -
         CATALOG(catalog.name)
/*
//STEP2 EXEC PLIXCLG
//PLI.SYSIN  DD  *
 CRR1:   PROC OPTIONS(MAIN);
         DCL NOS FILE RECORD OUTPUT DIRECT KEYED ENV(VSAM),
             CARD CHAR(80),
             NAME CHAR(20) DEF CARD,
             NUMBER CHAR(2) DEF CARD POS(21),
             IOFIELD CHAR(20);
         ON ENDFILE (SYSIN) GO TO FINISH;
         OPEN FILE(NOS);
 NEXT:   GET FILE(SYSIN) EDIT(CARD)(A(80));
         PUT FILE(SYSPRINT) SKIP EDIT (CARD) (A);
         IOFIELD=NAME;
         WRITE FILE(NOS) FROM(IOFIELD) KEYFROM(NUMBER);
         GO TO NEXT;

 FINISH: CLOSE FILE(NOS);
 END CRR1;
/*
//GO.NOS DD DSN=PL1VSAM.AJC3.BASE,DISP=OLD
//GO.SYSIN DD *
ACTION,G.            12
BAKER,R.             13
BRAMLEY,O.H.         28
CHEESNAME,L.         11
CORY,G.              36
ELLIOTT,D.           85
FIGGINS,E.S.         43
HARVEY,C.D.W.        25
HASTINGS,G.M.        31
KENDALL,J.G.         24
LANCASTER,W.R.       64
MILES,R.             23
NEWMAN,M.W.          40
PITT,W.H.            55
ROLF,D.E.            14
SHEERS,C.D.          21
SURCLIFFE,M.         42
TAYLOR,G.C.          47
WILTON,L.W.          44
WINSTONE,E.M.        37
//
```

Figure 109. Defining and Loading a Relative Record Data Set (RRDS)

Updating a Relative Record Data Set

Figure 110 on page 263 shows an RRDS being updated. A DIRECT UPDATE file is used and new records are written by key. There is no need to check for the records being empty, because the empty records are not available under VSAM.

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

The PL/I file is associated with the data sets by a DD statement that specifies the DSNAME PL1VSAM.AJC3.BASE, the name given in the DEFINE CLUSTER command in Figure 109 on page 261.

At the end of the example, the DELETE command is used to delete the data set.

```
//OPT9#18  JOB
//STEP1    EXEC PLIXCLG
//PLI.SYSIN  DD  *
 ACR1:  PROC OPTIONS(MAIN);
               DCL NOS FILE RECORD KEYED ENV(VSAM),NAME CHAR(20),
                 (NEWNO,OLDNO) CHAR(2),CODE CHAR(1),IOFIELD CHAR(20),
                 BYTE CHAR(1) DEF IOFIELD;
        ON ENDFILE(SYSIN) GO TO PRINT;
        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;

 NEXT:  GET FILE(SYSIN) EDIT(NAME,NEWNO,OLDNO,CODE)
           (COLUMN(1),A(20),A(2),A(2),A(1));
        PUT FILE(SYSPRINT) SKIP EDIT (' ',NAME,'#',NEWNO,OLDNO,' ',CODE)
           (A(1),A(20),A(1),2(A(2)),X(5),2(A(1)));
        IF CODE='A' THEN
          WRITE FILE(NOS) KEYFROM(NEWNO) FROM(NAME);
        ELSE IF CODE='C' THEN
          DO;
            DELETE FILE(NOS) KEY(OLDNO);
            REWRITE FILE(NOS) KEYFROM(NEWNO) FROM(NAME);
          END;
        ELSE IF CODE='D' THEN
          DELETE FILE(NOS) KEY(OLDNO);
        ELSE PUT FILE(SYSPRINT) SKIP EDIT
             ('INVALID CODE: ',NAME)(A(15),A);
        GO TO NEXT;

 PRINT:  CLOSE FILE(NOS);
         PUT FILE(SYSPRINT) PAGE;
         OPEN FILE(NOS) SEQUENTIAL INPUT;
         ON ENDFILE(NOS) GO TO FINISH;

 NEXTIN:  READ FILE(NOS) INTO(IOFIELD) KEYTO(NEWNO);
          PUT FILE(SYSPRINT) SKIP EDIT(NEWNO,IOFIELD)(A(5),A);
          GO TO NEXTIN;
 FINISH:  CLOSE FILE(NOS);
 END ACR1;
/*
//GO.NOS     DD DSN=PL1VSAM.AJC3.BASE,DISP=OLD
//GO.SYSIN   DD  *
NEWMAN,M.W.          5640C
GOODFELLOW,D.T.      89  A
MILES,R.               23D
HARVEY,C.D.W.        29  A
BARTLETT,S.G.        13  A
CORY,G.                36D
READ,K.M.            01  A
PITT,W.H.              55
ROLF,D.F.              14D
ELLIOTT,D.           4285C
HASTINGS,G.M.          31D
BRAMLEY,O.H.         4928C
/*
//STEP3    EXEC PGM=IDCAMS,REGION=512K,COND=EVEN
//SYSPRINT DD SYSOUT=A
//SYSIN    DD   *
      DELETE -
          PL1VSAM.AJC3.BASE -
          CATALOG(catalog.name)
//
```

Figure 110. Updating an RRDS

CHAPTER 8. LIBRARIES OF DATA SETS

Within the IBM operating system, the terms "partitioned data set" and "library" are used synonymously to signify a type of data set that can be used for the storage of other data sets (usually programs in the form of source, object or load modules). A library must be stored on direct-access storage and be wholly contained in one volume. It contains independent, consecutively-organized, data sets, called members. Each member has a unique name, not more than 8 characters long, which is stored in a directory that is part of the library. All the members of one library must have the same data characteristics because only one data set label is maintained.

Members can be created individually until there is insufficient space left for a new entry in the directory, or until there is insufficient space for the member itself. Members can be accessed individually by specifying the member name.

DD statements or their conversational mode equivalent are used to create and access members.

Members can be deleted by means of the IBM utility program IEHPROGM. This deletes the member name from the directory so that the member can no longer be accessed; but the space occupied by the member itself cannot be used again unless the library is recreated using, for example, the IBM utility program IEBCOPY. An attempt to delete a member by using the DISP parameter of a DD statement will cause the whole data set to be deleted.

TYPES OF LIBRARY

The following types of library may be used with a PL/I program:

- The system program library SYS1.LINKLIB or its equivalent. This can contain all system processing programs such as compilers and the linkage editor.
- Private program libraries. These usually contain user-written programs. It is often convenient to create a temporary private library to store the load module output from the linkage editor until it is executed by a later job step in the same job. The library will be deleted at the end of the job. Private libraries are also used for automatic library call by the linkage editor and the loader.
- The system procedure library SYS1.PROCLIB or its equivalent. This contains the job control procedures that have been cataloged for your installation.

HOW TO USE A LIBRARY

The ways in which the libraries described above can be used are described in the following sections.

BY THE LINKAGE EDITOR OR LOADER

The output from the linkage editor is usually placed on a private program library.

The call library used as input to the linkage editor or loader (see also Chapter 3, "The Linkage Editor and the Loader" on page 65) can be SYS1.LINKLIB, a private program library, or a subroutine library.

In each case, the processing of directory entries is performed by the operating system.

When you are adding a member to a library, you must specify the member name as follows:

- When a single module is produced as output from the linkage editor, the member name can be specified as part of the data set name (see also "Creating a Library Member" on page 267).
- When more than one module is produced as output from the linkage editor, the member name for each module must be specified in the NAME option or the NAME control statement. The member name can not be specified as part of the data set name.

BY THE OPERATING SYSTEM

When you request the execution of a load module in an EXEC statement or CALL command, the operating system must be able to retrieve the load module from a library. For a CALL command, this library is specified explicitly or implicitly in the command. For an EXEC statement, the following rules apply.

The operating system will assume the load module is a member of SYS1.LINKLIB, and will search in the directory for that library for the name you have specified, unless you have also specified that the load module is in a private library, in one of the following ways.

If the load module has been added to the private library in a previous step of the job (usually a link-edit step) and the member name was specified as part of the data set name, then you can refer, in the EXEC statement, to the DD statement defining the library instead of specifying the load module name. The library must have been given the disposition PASS.

If the load module exists on the private library before the job starts, then you have several ways of defining the library.

You can define the library in a DD statement, with the ddname JOBLIB, immediately after the JOB statement. This library will be used in place of SYS1.LINKLIB for all the steps of the job. If any load module is not found on the private library, the system will then look for in on SYS1.LINKLIB.

You can define the library in a DD statement with the ddname STEPLIB, at any point in the job control procedure. The private library will be used in place of SYS1.LINKLIB, or any library specified in a JOBLIB DD statement, for the job step in which it appears (though it can also be "passed" to subsequent job steps in the normal way). If any load module is not found on the private library, the system will look for in on the library specified on SYS1.LINKLIB; any JOBLIB will be ignored. The STEPLIB DD statement can be used in a cataloged procedure.

Alternatively, if you specify SYS1.LINKLIB in the JOBLIB or STEPLIB DD statements, and then concatenate the private library to it, the private library will be used only if a load module cannot be first found on SYS1.LINKLIB.

BY YOUR PROGRAM

Libraries can be used directly by a PL/I program.

If you are adding a new member to a library, its directory entry will be made by the operating system when the associated file is closed, using the member name specified as part of the data set name.

If you are accessing a member of a library, its directory entry can be found by the operating system from the member name that you specify as part of the data set name.

More than one member of the same library can be processed by the same PL/I program, but only one such output file can be open at any one time. Different members are accessed by giving the member name in a DD statement.

CREATING A LIBRARY

To create a library include in your job step a DD statement containing the information given in Figure 111. The information required is similar to that for a consecutively-organized data set (see "Defining a Consecutive Data Set" on page 151) except for the SPACE parameter.

Information Required	Parameter of DD Statement
Type of device that will be used	UNIT=
Serial number of the volume that will contain the library	VOLUME=SER
Name of the library	DSNAME=
Amount of space required for the library	SPACE=
Disposition of the library	DISP=

Figure 111. Information Required When Creating a Library

SPACE PARAMETER

The SPACE parameter in a DD statement that defines a library must always be of the form:

```
SPACE=(units,(quantity,
      increment,directory))
```

Although you can omit the third term (increment), indicating its absence by a comma, the last term, specifying the number of directory blocks to be allocated, must always be present.

The amount of auxiliary storage required for a library depends on the number and sizes of the members to be stored in it and on how often members will be added or replaced. (Space occupied by deleted members is not released.) The number of directory blocks required depends on the number of members and the number of aliases. Although you can specify an incremental quantity in the SPACE parameter that will allow the operating system to obtain more space for the data set if necessary, both at the time of creation and when new members are added, the number of directory blocks is fixed at the time of creation and cannot be increased.

The number of directory entries that a 256-byte directory block can contain depends on the amount of user data included in the entries. The maximum length of an entry is 74 bytes, but the entries produced by the linkage editor vary in length between 34 bytes and 52 bytes, which is equivalent to between four and seven entries per block.

For example, the DD statement:

```
//PDS DD UNIT=3330,VOLUME=SER=3412,
//    DSNAME=ALIB,
//    SPACE=(CYL,(5,,10)),
//    DISP=(,CATLG)
```

requests the job scheduler to allocate 5 cylinders of the 3330 disk pack with serial number 3412 for a new partitioned data set name ALIB, and to enter this name in the system catalog. The last term of the SPACE parameter requests that part of the space allocated to the data set be reserved for ten directory blocks.

CREATING A LIBRARY MEMBER

The members of a library must have identical characteristics. Otherwise, you may subsequently have difficulty retrieving them. This is necessary because the volume table of contents (VTOC) will contain only one data set control block (DSCB) for the library and not one for each member. When using a PL/I program to create a member, the operating system creates the directory entry; you cannot place information in the user data field.

When creating a library and a member at the same time, the DD statement must include all the parameters listed under "Creating a Library" on page 266, (although you can omit the DISP parameter if the data set is to be temporary). The DSNAME parameter must include the member name in parentheses. For example, DSNAME=ALIB(MEM1) names the member MEM1 in the data set ALIB. If the member is placed in the library by the linkage editor, you can use the linkage editor NAME statement or the NAME compiler option instead of including the member name in the DSNAME parameter. You must also describe the characteristics of the member (record format, etc.) either in the DCB parameter or in your PL/I program; these characteristics will also apply to other members added to the data set.

When creating a member to be added to an existing library, you will not need the SPACE parameter; the original space allocation applies to the whole of the library and not to an individual member. Furthermore, you will not need to describe the characteristics of the member, since these are already recorded in the DSCB for the library.

To add two more members to a library in one job step, you must include a DD statement for each member, and you must close one file that refers to the library before you open another.

EXAMPLES

The use of the cataloged procedure PLIXC to compile a simple PL/I program and place the object module in a new library named EXLIB is shown in Figure 112 on page 268. The DD statement that defines the new library and names the object module overrides the DD statement SYSLIN in the cataloged procedure. (The PL/I program is a function procedure that, given two values in the form of the character string produced by the TIME built-in function, returns the difference in milliseconds.)

The use of the cataloged procedure PLIXCL to compile and link-edit a PL/I program and place the load module in the existing library HPU8.CCLM is shown in Figure 113 on page 268.

```
//OPT10#1 JOB
//TR        EXEC  PLIXC
//PLI.SYSLIN  DD  UNIT=SYSDA,DSNAME=HPU8.EXLIB(ELAPSE),
//       SPACE=(TRK,(1,,1)),DISP=(NEW,CATLG)
//PLI.SYSIN    DD  *
   ELAPSE:  PROC(TIME1,TIME2);
     DCL (TIME1,TIME2) CHAR(9),
         H1 PIC '99' DEF TIME1,
         M1 PIC '99' DEF TIME1 POS(3),
         MS1 PIC '99999' DEF TIME1 POS(5),
         H2 PIC '99' DEF TIME2,
         M2 PIC '99' DEF TIME2 POS(3),
         MS2 PIC '99999' DEF TIME2 POS(5),
         ETIME FIXED DEC(7);
     IF H2<H1 THEN H2=H2+24;
     ETIME=((H2*60+M2)*600000+MS2)-((H1*60+M1)*600000+MS1);
     RETURN(ETIME);
   END ELAPSE;
/*
```

Figure 112. Creating New Libraries for Compiled Object Modules

```
//OPT10#2  JOB
//TRLE       EXEC  PLIXCL
//PLI.SYSIN    DD  *
   MNAME:  PROC  OPTIONS(MAIN);
      .
      .
      .
      program
      .
      .
      .

   END MNAME;
/*
//LKED.SYSLMOD  DD  DSNAME=HPU8.CCLM(DIRLIST),DISP=OLD
```

Figure 113. Placing a Load Module in an Existing Library

```
//OPT10#3  JOB
//TREX     EXEC  PLIXCLG
//PLI.SYSIN  DD  *
  NMEM:  PROC OPTIONS(MAIN);
     DCL IN FILE RECORD SEQUENTIAL INPUT,
         OUT FILE RECORD SEQUENTIAL OUTPUT,
         IOFIELD CHAR(80) BASED(A);
     OPEN FILE(IN),FILE(OUT);
     ON ENDFILE(IN) GO TO FINISH;

   NEXT: READ FILE(IN) SET(A);
     PUT FILE(SYSPRINT) SKIP EDIT (IOFIELD) (A);
     WRITE FILE(OUT) FROM(IOFIELD);
     GO TO NEXT;
   FINISH: CLOSE FILE(IN),FILE(OUT);
  END NMEM;
//GO.OUT   DD  UNIT=SYSDA,DSNAME=HPU8.ALIB(NMEM),
//      DISP=(NEW,CATLG),SPACE=(TRK,(1,1,1)),
//      DCB=(RECFM=FB,BLKSIZE=3600,LRECL=80)
//GO.IN DD *
//  MEM&COLON.PROC OPTIONS(MAIN);
       /* this is a dummy library member*/
/*
```

Figure 114. Creating a Library Member in a PL/I Program

To use a PL/I program to add or delete one or more records within a member of a library, you must rewrite the entire member in another part of the library; this is rarely an economic proposition, since the space originally occupied by the member cannot be used again. You must use two files in your PL/I program, but both can be associated with the same DD statement. The program shown in Figure 115 updates the member created by the program in Figure 114; it copies all the records of the original member except those that contain only blanks.

```
//OPT10#4  JOB
//TREX     EXEC   PLIXCLG
//PLI.SYSIN  DD  *
  UPDTM: PROC OPTIONS(MAIN);
     DCL (OLD,NEW) FILE RECORD SEQUENTIAL,
         DATA CHAR(80);
     ON ENDFILE(OLD)  GO TO FINISH;
     OPEN FILE(OLD) INPUT,FILE(NEW) OUTPUT TITLE('OLD');

   NEXT: READ FILE(OLD) INTO(DATA);
         PUT FILE(SYSPRINT) SKIP EDIT (DATA) (A);
         IF DATA=' ' THEN GO TO NEXT;
         WRITE FILE(NEW) FROM(DATA);
         GO TO NEXT;

   FINISH: CLOSE FILE(OLD),FILE(NEW);
  END UPDTM;
/*
//GO.OLD DD  DSNAME=HPU8.ALIB(NMEM),DISP=(OLD,DELETE),
//       UNIT=SYSDA,VOL=SER=nnnnnn
```

Figure 115. Updating a Library Member

LIBRARY STRUCTURE

The structure of a library is illustrated in Figure 116 on page 271. The directory of a library is a series of records (entries) at the beginning of the data set; there is at least one directory entry for each member. Each entry contains a member name, the relative address of the member within the library, and a variable amount of user data. The entries are arranged in ascending alphameric order of member names.

A directory entry can contain up to 62 bytes of user data (information inserted by the program that created the member). An entry that refers to a member (load module) written by the linkage editor includes user data in a standard format, described in the systems manuals.

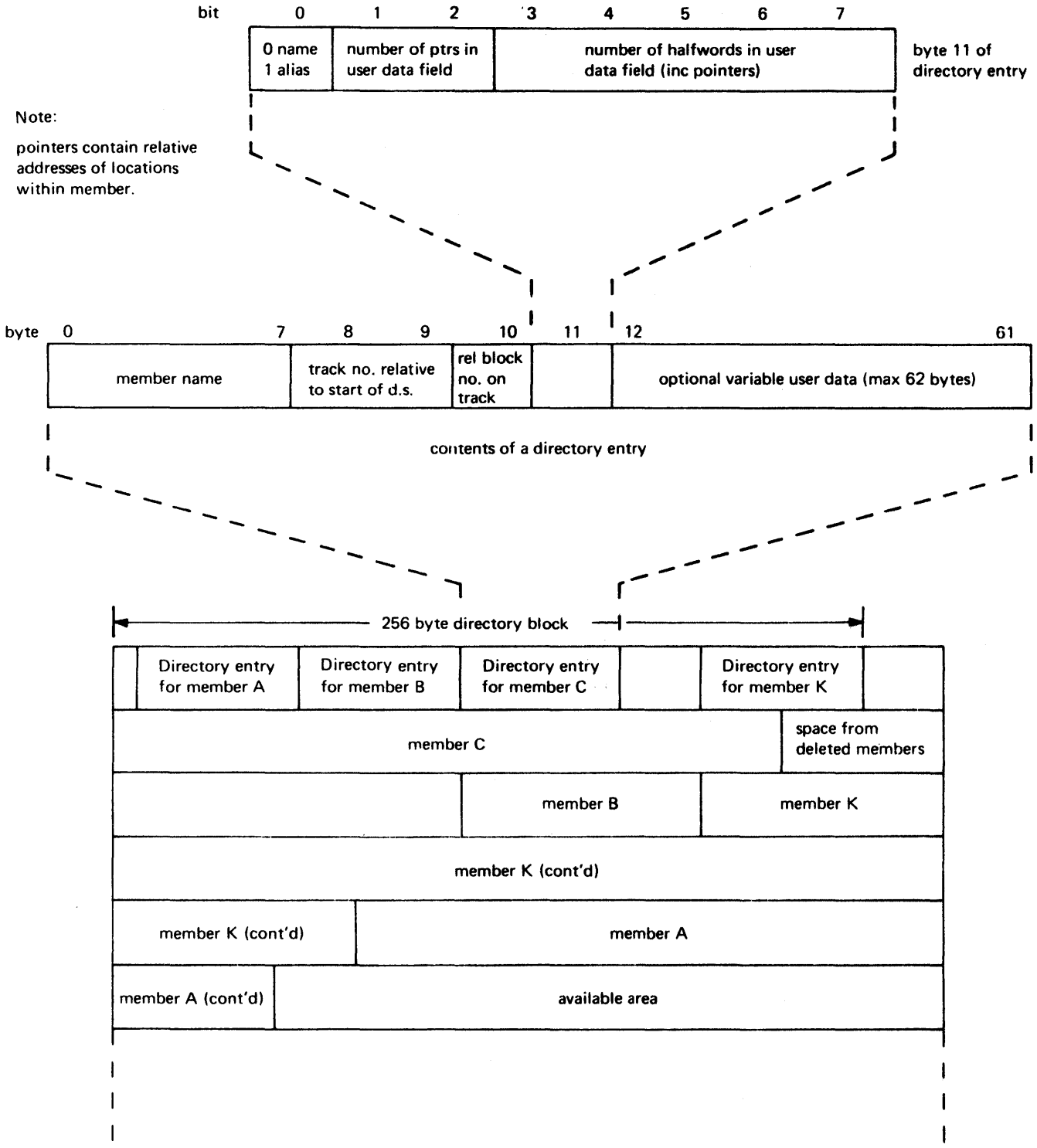


Figure 116. Structure of a Library

If you use a PL/I program to create a member, the operating system creates the directory entry for you and you cannot write any user data. However, you can use assembler language macro instructions to create a member and write your own user data; the method is described in the data management manuals.

Directory entries are stored in fixed-length blocks of 256 bytes, each containing a 2-byte count field specifying the number of active bytes in a block and as many complete entries as will fit into the remaining 254 bytes. The directory is in effect a sequential data set that contains fixed-length unblocked records, and can be read as such.

The program illustrated in Figure 117 demonstrates a method of extracting information from directory entries. The program lists the names of all the members of a library; the library must be defined, when the program is executed, in a DD statement with the name LINK.

```
//      EXEC PLIXCLG,PARM.PLI='MAR(1,72)'
//PLI.SYSIN DD *
MNAME: PROC OPTIONS(MAIN);

DCL LINK FILE RECORD SEQUENTIAL INPUT,
   1 DIRBLK,
    2 COUNT BIT(16),
    2 ENTRIES CHAR(254),
   1 ENTRY BASED(P),
    2 NAME CHAR(8),
    2 TTR  CHAR(3),
    2 INDIC,
     3 ALIAS  BIT(1),
     3 TTRS   BIT(2),
     3 USERCT BIT(5);
DCL LINK_EOF BIT(1) INIT('0'B) STATIC;

         ON ENDFILE(LINK) LINK_EOF = '1'B;
         READ FILE(LINK) INTO(DIRBLK);

         DO WHILE(¬LINK_EOF);
            DO UNSPEC(P) = UNSPEC(ADDR(ENTRIES))
                 REPEAT UNSPEC(UNSPEC(P) + 12 + 2*USERCT)
                 WHILE
                   (UNSPEC(P) <(UNSPEC(UNSPEC(ADDR(ENTRIES))+COUNT)));
            PUT FILE(SYSPRINT) SKIP LIST(NAME);
            END;
         READ FILE(LINK) INTO(DIRBLK);
         END;

         END MNAME;
//GO.LINK DD DSN=C.TEST.ASM,
//           DCB=(RECFM=U,BLKSIZE=256),
//           DISP=SHR
//
```

Figure 117. Listing Names of the Members of a Library

CHAPTER 9. CATALOGED PROCEDURES

This chapter describes the standard cataloged procedures supplied by IBM for use with the OS PL/I Optimizing Compiler, explains how to invoke them, and how to make temporary or permanent modifications to them.

A cataloged procedure is a set of job control statements stored in a system library, the procedure library. It includes one or more EXEC statements, each of which may be followed by one or more DD statements. You can retrieve the statements by naming the cataloged procedure in the PROC parameter of an EXEC statement in the input stream. When the operating system processes this EXEC statement, it replaces it in the input stream with the statements of the cataloged procedure.

The use of cataloged procedures saves time and reduces errors in coding frequently used sets of job control statements. If the statements in a cataloged procedure do not match your requirements exactly, you can easily modify them or add new statements for the duration of a job. It is recommended that each installation review these procedures and modify them to obtain the most efficient use of the facilities available and to allow for installation conventions.

INVOKING A CATALOGED PROCEDURE

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

```
//stepname EXEC PROC=PLIXC
```

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

```
//stepname EXEC PLIXC
```

When the operating system meets the name of a cataloged procedure in an EXEC statement, it extracts the statements of the cataloged procedure from the procedure library and substitutes them for the EXEC statement in the input job stream. If you include the parameter MSGLEVEL=1 in your JOB statement, the operating system will include the original EXEC statement in its listing, and will add the statements of the cataloged procedure. In the listing, cataloged procedure statements are identified by XX or X/ as the first two characters; X/ signifies a statement that has been modified for the current invocation of the cataloged procedure.

An EXEC statement identifies a **job step**, which can require either the execution of a program or the invocation of a cataloged procedure. A cataloged procedure includes one or more EXEC statements, which identify **procedure steps**. However, an EXEC statement in a cataloged procedure cannot invoke another cataloged procedure; it must request the execution of a program.

It may be necessary for you to modify the statements of a cataloged procedure for the duration of the job step in which it is invoked, either by adding DD statements or by overriding one or more parameters in the EXEC or DD statements. For example, cataloged procedures that invoke the compiler require the addition of a DD statement with the name SYSIN to define the data set containing the source statements. Also, whenever you use more than one standard link-edit procedure step in a job,

you must modify all but the first cataloged procedure that you invoke if you want to execute more than one of the load modules.

MULTIPLE INVOCATION OF CATALOGED PROCEDURES

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

The linkage editor always places a load module that it creates in the standard data set defined by the DD statement with the name SYSLMOD. In the absence of a linkage editor NAME statement (or the NAME compiler option), it uses the member name specified in the DSNAME parameter as the name of the module. In the standard cataloged procedures, the DD statement with the name SYSLMOD always specifies a temporary library named &&GOSET, and gives the load module the member name GO.

Consider what will happen if, for example, you use the cataloged procedure PLIXCLG twice in a job to compile, link edit, and execute two PL/I programs, and do not name each of the two load modules that will be created by the linkage editor. The linkage editor will name the first load module GO, as specified in the first DD statement with the name SYSLMOD. It will not be able to use the same name for the second load module since the first load module still exists in the library &&GOSET; it will allocate a temporary name to the second load module (a name that is not available to your program). Step GO of the cataloged procedure requests the operating system to initiate execution of the load module named in the first DD statement with the name SYSLMOD in the step LKED, that is, to execute the module named GO from the library &&GOSET. Consequently, the first load module will be executed twice and the second not at all.

To prevent this, use one of the following methods:

- Delete the library &&GOSET at the end of the step GO of the first invocation of the cataloged procedure by adding a DD statement with the syntax:

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

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

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

  and so on.

- Use the NAME compiler option to give a different name to each load module and change your job control statements to specify the execution of the load modules with these names.

MULTITASKING USING CATALOGED PROCEDURES

When you use a cataloged procedure to link edit a multitasking program, you must ensure that the load module includes the multitasking versions of the PL/I resident library subroutines. To enable you to select the appropriate library, the cataloged procedures that invoke the linkage editor and the loader include a symbolic parameter (&LKLBDSN) in the DSNAME parameter of the DD statement SYSLIB, which defines the data set to be used as the call library. This data set is described in "Automatic Call Library (SYSLIB)" on page 72. The default value of this symbolic parameter is SYS1.PLIBASE, which is the name of the nonmultitasking ("base") library.

To ensure that the multitasking library (SYS1.PLITASK) is searched before the base library, include the parameter LKLBDSN='SYS1.PLITASK' in the EXEC statement that invokes the cataloged procedure; for example:

```
//STEPA EXEC PLIXCLG,LKLBDSN='SYS1.PLITASK'
```

The DD statement SYSLIB is always followed in the standard cataloged procedures by another, unnamed, DD statement that includes the parameter DSNAME=SYS1.PLIBASE. The effect of this statement is to concatenate the base library with the multitasking library, if the latter is used; the base library can then be searched for any subroutine common to multitasking and nonmultitasking and therefore not included in the multitasking library. When the nonmultitasking library is selected, the second DD statement has no effect.

The use of the symbolic parameter &LKLBDSN means that for nonmultitasking programs, SYS1.PLIBASE is concatenated with itself. This has no effect other than a very small increase in job scheduling time, but does avoid the need for different cataloged procedures for link editing multitasking and nonmultitasking programs.

MODIFYING CATALOGED PROCEDURES

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

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

EXEC STATEMENT

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

- PARM applies to the first procedure step and nullifies any other PARM parameters.
- COND and ACCT apply to all the procedure steps.
- TIME and REGION apply to all the procedure steps and override existing values.

For example, the statement:

```
//stepname EXEC PLIXCLG,PARM='SIZE(MAX)',
      REGION=144K
```

invokes the cataloged procedure PLIXCLG, substitutes the option SIZE(MAX) for OBJECT and NODECK in the EXEC statement for procedure step PLI, and nullifies the PARM parameter in the EXEC statement for procedure step LKED; it also specifies a region size of 144K for all three procedure steps.

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

```
//stepname EXEC PROC=PLIXCLG,
     PARM='SIZE(MAX)',REGION.PLI=144K
```

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

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

```
//stepname EXEC PLIXCLG,PARM.LKED=
```

DD STATEMENT

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

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

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

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

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

Overriding DD statements must follow the EXEC statement that invokes the cataloged procedure in the same order as the corresponding DD statements of the cataloged procedure. DD statements that are being added must follow the overriding DD statements for procedure step in which they are to appear.

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

IBM-SUPPLIED CATALOGED PROCEDURES

The PL/I cataloged procedures supplied for use with the optimizing compiler are:

PLIXC Compile only

PLIXCL Compile and link edit

PLIXCLG Compile, link edit, and execute

PLIXLG Link edit and execute

PLIXCG Compile, load-and-execute

PLIXG Load-and-execute

The individual statements of the cataloged procedures are not fully described, since all the parameters are discussed elsewhere in this publication. These cataloged procedures do not include a DD statement for the input data set; you must always provide one. The example shown in Figure 118 illustrates the JCL statements you might use to invoke the cataloged procedure PLIXCLG to compile, link edit, and execute, a PL/I program.

```
//COLEGO    JOB
//STEP1     EXEC PLIXCLG
//PLI.SYSIN DD *
                .
                .
                .
    (insert here PL/I program to be
    compiled)
                .
                .
                .
/*
```

Figure 118. Invoking a Cataloged Procedure

No IBM-supplied cataloged procedure is provided to produce an object module on punched cards. You can temporarily modify any of the cataloged procedures that have a compile step to produce a punched card output; as example is shown in Figure 119.

```
//stepname     EXEC PLIXCLG,
//              PARM.PLI='OBJECT,DECK'
//PLI.SYSPUNCH DD SYSOUT=B
//PLI.SYSIN    DD ...
               .
               .
               .
```

Figure 119. Modifying a Cataloged Procedure to Produce a Punched Card Output

COMPILE ONLY (PLIXC)

This cataloged procedure, shown in Figure 120, includes only one procedure step, in which the options specified for the compilation are OBJECT and NODECK. (IEL0AA is the symbolic name of the compiler.) In common with the other cataloged procedures that include a compilation procedure step, PLIXC does not include a DD statement for the input data set; you must always supply an appropriate statement with the qualified ddname PLI.SYSIN.

The OBJECT option causes the compiler to place the object module, in a syntax suitable for input to the linkage editor, in the standard data set defined by the DD statement with the name SYSLIN. This statement defines a temporary data set name &&LOADSET on a magnetic-tape or direct-access device; if you want to retain the object module after the end of your job, you must substitute a permanent name for &&LOADSET (that is, a name that does not commence &&) and specify KEEP in the appropriate DISP parameter for the last procedure step in which the data set is used.

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

The SPACE parameter allows an initial allocation of 250 eighty-byte records and, if necessary, 15 further allocations of 100 records (a total of 1750 records, which should suffice for most applications).

```
//PLIXC   PROC
//PLI     EXEC PGM=IEL0AA,PARM='OBJECT,NODECK,COMPILE',REGION=128K
//SYSPRINT  DD SYSOUT=A
//SYSLIN  DD DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSSQ,
//           SPACE=(80,(250,100))
//SYSUT1  DD DSN=&&SYSUT1,UNIT=SYSDA,DCB=BLKSIZE=1024
//           SPACE=(1024,(200,50),,CONTIG,ROUND)
```

Figure 120. Cataloged Procedure PLIXC

COMPILE AND LINK-EDIT (PLIXCL)

This cataloged procedure, shown in Figure 121 on page 279, includes two procedure steps: PLI, which is identical with cataloged procedure PLIXC, and LKED, which invokes the linkage editor (symbolic name IEWL) to link edit the object module produced in the first procedure step.

Input data for the compilation procedure step requires the qualified ddname PLI.SYSIN. The COND parameter in the EXEC statement LKED specifies that this procedure step should be bypassed if the return code produced by the compiler is greater than 9 (that is, if a severe or unrecoverable error occurs during compilation).

The DD statement with the name SYSLIB specifies the PL/I resident library, from which the linkage editor will obtain appropriate modules for inclusion in the load module. The linkage editor always places the load modules it creates in the standard data set defined by the DD statement with the name SYSLMOD. This statement in the cataloged procedure specifies a new temporary library &&GOSET, in which the load module will be placed and given the member name GO (unless you specify the NAME compiler option for the compiler procedure step). In specifying a temporary library, the cataloged procedure assumes that you

will execute the load module in the same job; if you want to retain the module, you must substitute your own statement for the DD statement with the name SYSLMOD.

```
//PLIXCL  PROC LKLBDSN='SYS1.PLIBASE'
//PLI     EXEC PGM=IEL0AA,PARM='OBJECT,NODECK',REGION=128K
//SYSPRINT  DD SYSOUT=A
//SYSLIN  DD DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSSQ,
//           SPACE=(80,(250,100))
//SYSUT1  DD DSN=&&SYSUT1,UNIT=SYSDA,DCB=BLKSIZE=1024,
//           SPACE=(1024,(200,50),,CONTIG,ROUND),
| //LKED    EXEC PGM=IEWL,PARM='XREF,LIST',COND=(9,LT,PLI),REGION=256K
//SYSLIB  DD DSN=&LKLBDSN,DISP=SHR
//         DD DSN=SYS1.PLIBASE,DISP=SHR
//SYSLMOD DD DSN=&&GOSET(GO),DISP=(MOD,PASS),UNIT=SYSDA,
//           SPACE=(1024,(50,20,1))
//SYSUT1  DD DSN=&&SYSUT1,UNIT=SYSDA,DCB=BLKSIZE=1024,
//           SPACE=(1024,(200,50),,CONTIG,ROUND),
//SYSPRINT  DD SYSOUT=A
//SYSLIN  DD DSN=&&LOADSET,DISP=(OLD,DELETE)
//         DD DDNAME=SYSIN
//SYSIN   DD DUMMY
```

Figure 121. Cataloged Procedure PLIXCL

```
//PLIXCLG  PROC LKLBDSN='SYS1.PLIBASE'
| //PLI     EXEC PGM=IEL0AA,PARM='OBJECT,NODECK',REGION=128K
//SYSPRINT  DD SYSOUT=A
//SYSLIN  DD DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSSQ,
//           SPACE=(80,(250,100))
//SYSUT1  DD DSN=&&SYSUT1,UNIT=SYSDA,DCB=BLKSIZE=1024,
//           SPACE=(1024,(200,50),,CONTIG,ROUND),
| //LKED    EXEC PGM=IEWL,PARM='XREF,LIST',COND=(9,LT,PLI),REGION=256K
//SYSLIB  DD DSN=&LKLBDSN,DISP=SHR
//         DD DSN=SYS1.PLIBASE,DISP=SHR
//SYSLMOD DD DSN=&&GOSET(GO),DISP=(MOD,PASS),UNIT=SYSDA,
//           SPACE=(1024,(50,20,1))
//SYSUT1  DD DSN=&&SYSUT1,UNIT=SYSDA,SPACE=(1024,(200,20)),
//           DCB=BLKSIZE=1024
//SYSPRINT  DD SYSOUT=A
//SYSLIN  DD DSN=&&LOADSET,DISP=(OLD,DELETE)
//         DD DDNAME=SYSIN
//SYSIN   DD DUMMY
//GO      EXEC PGM=*.LKED.SYSLMOD,COND=((9,LT,PLI),(9,LT,LKED)),
//           REGION=100K
//SYSPRINT  DD SYSOUT=A
```

Figure 122. Cataloged Procedure PLIXCLG

The last statement, DDNAME=SYSIN, illustrates how to concatenate a data set defined by a DD statement with the name SYSIN with the primary input (SYSLIN) to the linkage editor. You could place linkage editor control statements in the input stream by this means, as described in "Primary Input (SYSLIN)" on page 71.

COMPILE, LINK-EDIT AND EXECUTE (PLIXCLG)

This cataloged procedure, shown in Figure 122 on page 279, includes three procedure steps, PLI and LKED, which are identical with the two procedure steps of PLIXCL, and GO, in which the load module created in the step LKED is executed. The third procedure step will be executed only if no severe or unrecoverable errors occur in the preceding procedure steps.

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

LINK-EDIT AND EXECUTE (PLIXLG)

This cataloged procedure, shown in Figure 123, includes two procedure steps, LKED and GO, which are similar to the procedure steps of the same names in PLIXCLG.

```
//PLIXLG  PROC LKLBDSN='SYS1.PLIBASE'
//LKED    EXEC PGM=IEWL,PARM='XREF,LIST',REGION=256K
//SYSLIB  DD DSN=&LKLBDSN,DISP=SHR
//        DD DSN=SYS1.PLIBASE,DISP=SHR
//SYSLMOD DD DSN=&&GOSET(GO),DISP=(MOD,PASS),UNIT=SYSDA,
//           SPACE=(1024,(50,20,1))
//SYSUT1  DD DSN=&&SYSUT1,UNIT=SYSDA,SPACE=(1024,(200,20)),
//           DCB=BLKSIZE=1024
//SYSPRINT  DD SYSOUT=A
//SYSLIN  DD DDNAME=SYSIN
//SYSIN   DD DUMMY
//GO      EXEC PGM=*.LKED.SYSLMOD,COND=(9,LT,LKED),REGION=100K
//SYSPRINT  DD SYSOUT=A
```

Figure 123. Cataloged Procedure PLIXLG

In the procedure step LKED, the DD statement with the name SYSLIN does not define a data set, but merely refers the operating system to the DD statement SYSIN, which you must supply with the qualified ddname LKED.SYSIN. This DD statement defines the data set from which the linkage editor will obtain its primary input. Execution of the procedure step GO is conditional on successful execution of the procedure step LKED only.

COMPILE, LOAD, AND EXECUTE (PLIXCG)

This cataloged procedure, shown in Figure 124, achieves the same results as PLIXCLG but uses the loader instead of the linkage editor. However, instead of using three procedure steps (compile, link edit, and execute), it has only two (compile, and load-and-execute). In the second procedure step, the loader program is executed; this program processes the object module produced by the compiler and executes the resultant executable program immediately. Input data for the compilation procedure step requires the qualified ddname PLI.SYSIN.

The REGION parameter of the EXEC statement GO specifies 100K bytes. Since the loader requires about 17K bytes of main storage, there are about 83K bytes for your program; if this is likely to be insufficient, you must modify the REGION parameter. The use of the loader imposes certain restrictions on your PL/I program; before using this cataloged procedure, see "Loader" on page 66, which explains how to use the loader.

```
//PLIXCG  PROC LKLBDSN='SYS1.PLIBASE'
//PLI     EXEC PGM=IELOAA,PARM='OBJECT,NODECK',REGION=128K
//SYSPRINT  DD SYSOUT=A
//SYSLIN  DD DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSSQ,
//           SPACE=(80,(250,100))
//SYSUT1  DD DSN=&&SYSUT1,UNIT=SYSDA,DCB=BLKSIZE=1024,
//           SPACE=(1024,(200,50),,CONTIG,ROUND)
//GO      EXEC PGM=LOADER,PARM='MAP,PRINT',REGION=324K,
//           COND=(9,LT,PLI)
//SYSLIB  DD DSN=&LKLBDSN,DISP=SHR
//        DD DSN=SYS1.PLIBASE,DISP=SHR
//SYSLIN  DD DSN=&&LOADSET,DISP=(OLD,DELETE)
//SYSLOUT DD SYSOUT=A
//SYSPRINT  DD SYSOUT=A
```

Figure 124. Cataloged Procedure PLIXCG

LOAD AND EXECUTE (PLIXG)

This cataloged procedure, shown in Figure 125, achieves the same results as PLIXLG but uses the loader instead of the linkage editor. However, instead of using two procedure steps (link edit and execute), it has only one. In this procedure step, the loader program is executed. This program processes and executes an object module placed in the data set defined by a DD statement with the name SYSLIN; you must supply this statement with the qualified name GO.SYSLIN.

```
//PLIXG   PROC LKLBDSN='SYS1.PLIBASE'
//GO      EXEC PGM=LOADER,PARM='MAP,PRINT',REGION=324K
//SYSLIB  DD DSN=&LKLBDSN,DISP=SHR
//        DD DSN=SYS1.PLIBASE,DISP=SHR
//SYSLOUT DD SYSOUT=A
//SYSPRINT DD SYSOUT=A
```

Figure 125. Cataloged Procedure PLIXG

The REGION parameter of the EXEC statement GO specifies 100K bytes. Since the loader requires about 17K bytes of main storage, there are about 83K bytes for your program; if this is likely to be insufficient, you must modify the REGION parameter. The use of the loader imposes certain restrictions on your PL/I program; before using this cataloged procedure, see "Loader" on page 66, which explains how to use the loader.

CHAPTER 10. PROGRAM CHECKOUT

Program checkout is the application of diagnostic and test processes to a program. You should give adequate attention to program checkout during the development of a program so that:

- A program becomes fully operational after the fewest possible test runs, thereby minimizing the time and cost of program development.
- A program is proved to have fulfilled all the design objectives before it is released for production work.
- A program has complete and clear documentation to enable both operators and program maintenance personnel to use and maintain the program without assistance from the original programmer.

The data used for the checkout of a program should be selected to test all parts of the program. While the data should be sufficiently comprehensive to provide a thorough test of the program, it is easier and more practical to monitor the behavior of the program if the volume of data is kept to a minimum.

CONVERSATIONAL PROGRAM CHECKOUT

The optimizing compiler can be used in conversational mode when writing and testing programs at the terminal. The conversational features are available to users where the TSO (Time Sharing Option) facilities of the operating system or CMS (Conversational Monitor System) are present. The conversational facilities enable you to enter a PL/I program from a terminal, through which you will receive diagnostic messages for the compilation. You can also communicate with the program during execution using PL/I files associated with the terminal. Thus a PL/I program can be checked out during its construction, thereby saving a substantial amount of elapsed time that can occur between test compilation and execution runs in batched processing. Attention on-units can be incorporated to put out debugging information when an attention interrupt is caused from the terminal.

The PL/I program is entered and processed using the commands and features described in the: OS PL/I Optimizing Compiler: TSO User's Guide, and the OS PL/I Optimizing Compiler: CMS User's Guide.

COMPILE-TIME CHECKOUT

At compile time, both the preprocessor and the compiler can produce diagnostic messages and listings according to the compiler options selected for a particular compilation. The listings and the associated compiler options are discussed in "Compiler Options" on page 11. The diagnostic messages produced by the optimizing compiler are identified by a number prefixed "IEL." These diagnostic messages are available in both a long form and a short form. The long messages are designed for reproduction at a terminal when the compiler is being used in a TSO environment. The short messages are obtained by specifying the SMESSAGE compiler option. Each message is reproduced in the publication: OS PL/I Optimizing Compiler Messages. This publication includes explanatory notes, examples, and any action to be taken.

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 appreciate that the compiler, in making an assumption as to the intended meaning of any erroneous statement in the source program, can introduce a further, perhaps more severe, error which in turn can produce yet another error, and so on. When this occurs, the result is that the compiler produces a number of diagnostic messages which are all caused either directly or indirectly by the one 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 will suppress optimization unconditionally, and the remaining options will suppress compilation, link editing, and execution should the appropriate error conditions be detected.

The NOSYNTAX option specified with the severity level "W," "E," or "S" will cause compilation of the output from the PL/I preprocessor, if used, to be suppressed prior to the syntax-checking stage should the preprocessor issue diagnostic messages at or above the severity level specified in the option.

The NOCOMPILE option specified with the severity level "W," "E," or "S" will cause compilation to be suppressed after the syntax-checking stage if syntax checking or preprocessing causes the compiler to issue diagnostic messages at or above the severity level specified in the option.

LINKAGE EDITOR CHECKOUT

When using the linkage editor, check particularly that any required overlay structuring and incorporation of additional object and load modules have been performed correctly. Diagnostic messages produced by the linkage editor are prefixed "IEW." These messages are fully documented in the publication: OS Linkage Editor and Loader Messages.

When checking the processing performed by the linkage editor, refer to the module map produced by the linkage editor showing the structure of the load module. The module map names the modules that have been incorporated into the program. The compiler produces an external symbol dictionary (ESD) listing if requested by the ESD option. The ESD listing indicates the external names that the linkage editor is to resolve in order to create a load module. The linkage editor is described in Chapter 3, "The Linkage Editor and the Loader" on page 65.

EXECUTION-TIME CHECKOUT

At execution time, errors can occur in a number of different operations associated with running a program. For instance, an error in the use of a job control statement can cause a job to fail. Most errors that can be detected are indicated by a diagnostic message. The diagnostic messages for errors detected at execution time are also listed in the messages publication for this compiler and identified by the prefix "IBM." The messages are always printed on the SYSPRINT file.

A failure in the execution of a PL/I program could be caused by one of the following:

- Logical errors in source programs.
- Invalid use of PL/I.
- Unforeseen errors.
- Operating error.
- Invalid input data.
- Unidentified program failure.
- A compiler or library subroutine failure.
- System failure.

LOGICAL ERRORS IN SOURCE PROGRAMS

Logical errors in source programs can often be difficult to detect. Such errors can sometimes cause a compiler or library failure to be suspected. The more common errors are the failure to convert correctly from arithmetic data, incorrect arithmetic operations and string manipulation operations, and failure to match data lists with their format lists.

INVALID USE OF PL/I

It is possible that a misunderstanding of the language or the failure to provide the correct environment for using PL/I, results in an apparent failure of a PL/I program. For example, the use of uninitialized variables, the use of controlled variables that have not been allocated, reading records into incorrect structures, the misuse of array subscripts, the misuse of pointer variables, conversion errors, incorrect arithmetic operations, and incorrect string manipulation operations can cause this type of failure.

UNFORESEEN ERRORS

If an error is detected during execution of a PL/I program in which no on-unit is provided to terminate execution or attempt recovery, the job will be terminated abnormally. However, the status of a program executed in a batch-processing environment, at the point where the error occurred, can be recorded by the use of an ERROR on-unit that contains the statements:

```
ON ERROR BEGIN;
ON ERROR SYSTEM;
PUT DATA;
END;
```

The statement ON ERROR SYSTEM; contained in the on-unit ensures that further errors caused by attempting to transmit uninitialized variables do not result in a permanent loop.

OPERATING ERROR

A job could fail because of an operating error, such as running a job twice so that a data set becomes overwritten or erroneously deleted. Other operating errors include getting card decks into the wrong order and the failure to give operators correct instructions for running a job.

INVALID INPUT DATA

A program should contain checks to ensure that any incorrect input data is detected before it can cause the program to fail.

Use the COPY option of the GET statement if you wish to check values obtained by stream-oriented input. The values will be listed on the file named in the COPY option. If no file name is given, SYSPRINT is assumed.

UNIDENTIFIED PROGRAM FAILURE

In most circumstances, an unidentified program failure should not occur when using the optimizing compiler. Exceptions to this could include the following:

- When the program is executed in conjunction with non-PL/I modules, such as FORTRAN or COBOL.
- When the program obtains, by means of record-oriented transmission, incorrect values for use in label, entry, locator, and file variables.
- Errors in job control statements, particularly in defining data sets.

If execution of a program terminates abnormally without an accompanying PL/I execution-time diagnostic message, it is probable that the error that caused the failure also inhibited the production of a message. In this situation, it is still possible to check the PL/I source program for errors that could result in overwriting areas of the main storage region that contain executable instructions, particularly the communications region, which contains the address tables for the execution-time error-handling routine. These errors may also be present in modules compiled by the checkout compiler with NODIAGNOSE and COMPATIBLE and executed in conjunction with the modules produced by the optimizing compiler. The types of PL/I program that might cause the main storage to be overwritten erroneously are:

- Assignment of a value to a non-existent array element. For example:

```
DCL ARRAY(10);
.
.
.
DO I = 1 TO 100;
ARRAY(I) = VALUE;
```

  To detect this type of error in a module compiled by the optimizing compiler, enable the SUBSCRIPTRANGE condition. For each attempt to access an element outside the declared range of subscript values, the SUBSCRIPTRANGE condition will be raised. If there is no on-unit for this condition, a diagnostic message will be printed and the ERROR condition raised. This facility, although expensive in execution time and storage space, is a valuable program-checkout aid.

- The use of incorrect locator values for locator (pointer and offset) variables. This type of error is possible if a locator value is obtained by means of record-oriented transmission. Check that locator values created in a program, transmitted to a data set, and subsequently retrieved for use in another program, are valid for use in the second program.

An error could also be caused by attempting to free a non-based variable. This could be caused by freeing a based variable when its qualifying pointer value has been changed. For example:

```
DCL A STATIC,B BASED (P);
ALLOCATE B;
P = ADDR(A);
FREE B;
```

- The use of incorrect values for label, entry, and file variables. Errors similar to those described above for locator values are possible for label, entry, and file values that are transmitted and subsequently retrieved.
- The use of the SUBSTR pseudovariable to assign a string to a position beyond the maximum length of the target string. For example:

```
DCL X CHAR(3);
I=3
SUBSTR(X,2,I) = 'ABC';
```

The STRINGRANGE condition can be used to detect this type of error in a module compiled by the optimizing compiler.

COMPILER OR LIBRARY SUBROUTINE FAILURE

If you are absolutely convinced that the failure is caused by a compiler failure or a library subroutine failure, you should notify your management, who will initiate the appropriate action to correct the error. This could mean calling in IBM personnel for programming support to rectify the problem. Before calling IBM for programming support, refer to the instructions for providing the correct information to be used in diagnosing the problem. These instructions are given in Appendix B, "Requirements For Problem Determination And APAR Submission" on page 400. Meanwhile, you can attempt to find an alternative way to perform the operation that is causing the trouble. A bypass is often feasible, since the PL/I language frequently provides an alternative method of performing a given operation.

SYSTEM FAILURE

System failures include machine malfunctions and operating system errors. These failures should be identified to the operator by a system message.

STATEMENT NUMBERS AND TRACING

The compiler FLOW option provides a valuable program-checkout aid. The FLOW(n,m) option creates a table of the numbers of the last "n" branch-out and branch-in statements, and the last "m" procedures and on-units to be entered. (A 'branch-out' statement is a statement that transfers control to a statement other than that which immediately follows it, such as a GOTO statement. A branch-in statement is a statement that receives control from a statement other than that which immediately precedes it, such as a PROCEDURE, ENTRY, or any other labeled statement.) The figure you choose for 'n' should be large enough to provide a usable trace of the flow of control through the program. Alternatively, if you do not specify n and m explicitly, defaults for the FLOW option will be used.

The trace table can be obtained by any of the methods described below.

The trace is printed whenever an on-unit with the SNAP option is encountered. It gives both the statement numbers and the names of the containing procedures or on-units. For example, an ERROR

on-unit that results in both the listing of the program variables and the statement number trace can be included in a PL/I program as follows:

```
ON ERROR SNAP BEGIN;
ON ERROR SYSTEM;
PUT DATA;
END;
```

A flow trace can be specified as part of the output from the PL/I dump facility PLIDUMP, discussed later in "Dumps" on page 288.

If the OPTIMIZE and REORDER options are used when compiling, the numbers produced by COUNT and FLOW may be useful but they are not accurate.

DYNAMIC CHECKING FACILITIES

It is possible for a syntactically-correct program to produce incorrect results without raising any PL/I error conditions. This can be attributed to the use of incorrect logic in the PL/I source program or to invalid input data. Detection of such errors from the resultant output (if any) can be a difficult task. It is sometimes helpful to have a record of each of the values assigned to a variable, particularly label, entry, loop control, and array subscript variables. This can be obtained by using the CHECK prefix option. Note that, unless care is exercised, the indiscriminate use of the facilities described below will result in a flood of unwanted and unusable printout.

A CHECK prefix option can specify program variables in a list. Whenever a variable that has been included in a check-list is assigned a new value, the CHECK condition is raised. The implicit action for the CHECK condition is to print the name and new value of the variable that caused the CHECK condition to be raised. An example of a CHECK prefix option list is:

```
(CHECK(A,B,C,L)):/* CHECKOUT PREFIX LIST */
    TEST: PROCEDURE OPTIONS(MAIN);
          DECLARE A etc.,
          .
          .
          .
```

If the CHECK condition is to be raised for all the variables used in a program, the CHECK prefix option can be more simply specified without a list of items. For example:

```
(CHECK): TEST: PROCEDURE;
```

CONTROL OF CONDITIONS

During execution of a PL/I object program, a number of conditions can be raised, either as a result of program-defined action, or as a result of exceeding a hardware limitation. PL/I contains facilities for detecting such conditions. These facilities can be used to determine the circumstances of an unexpected interrupt, perform a recovery operation, and permit the program to continue to run. Alternatively, the facilities can be used to detect conditions raised during normal processing, and initiate program-defined actions for the condition. Note that some of the PL/I conditions are enabled by default, some cannot be disabled, and others have to be enabled explicitly in the program. Refer to the OS and DOS PL/I Language Reference Manual for a full description of each condition.

Note that the SIGNAL statement can be used to raise any of the PL/I conditions. Such use permits any on-units in the program to be tested during debugging.

The implicit action for the ERROR condition for which there is no on-unit, is, in batched processing, to raise the FINISH condition, and in interactive processing, to give control to the terminal. The FINISH condition is also raised for the following:

- When a SIGNAL FINISH statement is executed.
- When a PL/I program completes execution normally.
- On completion of an ERROR on-unit that does not return control to the PL/I program by means of a GOTO statement.
- When a STOP statement is executed or when an EXIT statement is executed in a major task.

The implicit action for the FINISH condition in batched processing is to terminate the task, and, in interactive processing, to give control to the terminal.

USE OF THE PL/I PREPROCESSOR IN PROGRAM CHECKOUT

During program checkout, it is often necessary to use a number of the PL/I conditions (and the on-units associated with them) and subsequently to remove them from the program when it is found to be satisfactory. The PL/I preprocessor can be used to include program-checkout statements from the source statement library. When the program is fully operational, the %INCLUDE statement can be removed, and the resultant object program compiled for execution.

PL/I program checkout statements would include both the enabling of any conditions that are disabled by default and the provision of the appropriate on-units. The %INCLUDE statement that causes the inclusion of the program checkout statements would usually be placed after any on-units that must remain in the program permanently in order to cancel their effect during program checkout.

CONDITION CODES

Condition codes can indicate more precisely what type of error has occurred where a condition can be raised by more than one error. For example, the ERROR condition can be raised by a number of different errors, each of which is identified by a condition code. You can obtain the condition code by using the ONCODE built-in function in the on-unit. The condition codes are described in the OS and DOS PL/I Language Reference Manual.

DUMPS

Should the checks given above fail to reveal the cause of the error, it may be necessary to obtain a printout, or dump, of all or part of the storage used by the program. The OS PL/I Optimizing Compiler allows you to obtain an execution-time dump only by calling PLIDUMP.

See OS PL/I Optimizing Compiler: Execution Logic, for information about the organization of the object programs produced by the optimizing compiler, and how to interpret the PLIDUMP outputs.

A DD statement with the name PLIDUMP or PL1DUMP must be supplied to define the data set for the dump.

The data set defined by the PLIDUMP DD statement must have DSORG=PS specified or assumed by default, and must have one of the following attributes:

- It must be allocated to SYSOUT.

- It must be allocated to the terminal or unit-record device.
- DISP=MOD must be specified.

The page size of the PLIDUMP output is taken from the PAGESIZE field of PLITABS.

To obtain a formatted PL/I dump, you must call PLIDUMP. PLIDUMP can be invoked with two optional arguments. The format of the CALL PLIDUMP statement is:

```
CALL PLIDUMP[(options-list
     [,user-identification])];
```

The first argument, **option-list**, is a character-string expression that specifies the type of information to be included in the dump. The options-list may include the following:

T To request a trace of active procedures, begin blocks, on-units, and library modules.

NT To suppress the output produced by T above.

F To request a complete set of attributes for all files that are open, and the contents of the buffers used by the files.

NF To suppress the output produced by F above.

S To request the termination of the program after the completion of the dump. Note: The FINISH condition is not raised.

C To request continuation of execution after completion of the dump.

H To request a hexadecimal dump of the storage used by the program.

NH To suppress the hexadecimal dump.

B If T is specified, to produce a separate hexadecimal dump of control blocks such as the TCA and the DSA chain that are used in the trace analysis. If F is specified, to produce a separate hexadecimal dump of control blocks used in the file analysis, such as the FCB.

NB To suppress hexadecimal dumps of control blocks.

A To request information relevant to all tasks in a multitasking program.

E To request that an exit be made from the current task of a multitasking program and that execution of the program continues after the completion of the requested dump.

O To request information relevant only to the current task in a multitasking program.

The defaults assumed for the above options not specified explicitly are:

```
T F C A NH NB
```

The second argument, **user-identification**, specifies the identification to be printed at the head of the dump. It can be a character-string expression of up to 90 characters or a decimal constant.

EXAMPLE

An example of the CALL PLIDUMP statement is:

```
CALL PLIDUMP ('TFCNH',
     'DUMP AFTER READ');
```

TRACE INFORMATION

Trace information produced by PLIDUMP includes a trace through all the active DSAs. (DSAs will be present for compiled blocks, such as procedures and on-units, and for library routines.) For on-units, the dump gives the values of any condition built-in functions that could be used in the on-unit, regardless of whether the on-unit actually used the condition built-in function. If a hexadecimal dump is also requested, the trace information will also include:

- The address of each DSA (Dynamic Storage Area).
- The address of the TCA (Task Communications Area).
- The contents of the registers on entry to the PL/I error-handler module (IBMCERR).
- The PSW or the address from which the PL/I error handler module (IBMBERR) was invoked.
- The addresses of the library module DSAs back to the most recently-used compiled code DSA.

DSAs and the TCA are described in the OS PL/I Optimizing Compiler: Execution Logic. A table of statement numbers indicating the flow of control through the program is produced if the FLOW option is in effect.

FILE INFORMATION

File information produced by PLIDUMP includes the attributes of all open files, and the contents of all buffers that are accessible to the dump routine. The information is given in EBCDIC notation, and, in hexadecimal notation also. The address and contents of the FCB are then printed. For varying length records, the RECSIZE is the length of the last processed record.

HEXADECIMAL DUMP

To use a hexadecimal storage dump, you should understand object program organization. The hexadecimal dump is a dump of the region of storage containing the program. The dump is given as three columns of printed output. The left-hand and middle columns contain the contents of storage in hexadecimal notation. The third column contains a EBCDIC translation of the first two columns. For hexadecimal characters that cannot be represented by a printable EBCDIC character, a period is printed.

EXECUTION-TIME RETURN CODES

It is possible to pass a return code from a PL/I program to the program that invoked the PL/I program. For example, if the PL/I program is invoked by the operating system, a return code can be passed either for examination in a subsequent job step if execution of that step is conditional upon the value of the code returned, or merely to indicate conditions that were encountered during execution. Conditional execution of a job step is determined by use of the COND parameter of the JOB or EXEC statement.

The return code generated by a PL/I program consists of 2 elements. One element is specified if the program calls PLIRETC

or is set to zero by default. The other is specified by the program management routines of the PL/I library and indicates the way in which your program terminated. Unless an error is detected which prevents the PL/I program management routines from operating correctly, the two elements are added together to form a total in which the thousands digit indicates the way in which your program terminated and the hundreds, tens, and units are set by your program when PLIRETC is called, and can be used to allow conditional execution of the next step or for any other purpose you require.

When a PL/I program calls PLIRETC, the argument (return code value) can be either a constant or a variable with the attributes FIXED BINARY(31,0). If a return code greater than 999 is specified, the return code is set to 999 and a diagnostic message is issued.

The meaning of the thousands digit generated by the PL/I program management routines is as follows:

0000 Normal termination.

1000 STOP or EXIT statement, or a call to PLIDUMP with the S or E option, or insufficient storage in the ISA.

2000 ERROR condition raised and program terminated without return from ERROR or FINISH on-unit.

3000 Abends in the 3000-3999 range can be issued by a user-written IBMBEER module. For further details, see "The Abend Facility" on page 292.

4000 Error prevented program management routines from functioning correctly. In this situation the remaining digits are used to further identify the error as shown below, and any set by a call to PLIRETC are ignored.

4004 Code returned if the PRV (pseudo register vector) is too large. Suggested Corrective Action:

Reduce the number of files and/or controlled variables in the program.

4008 Code returned if PL/I program has no main procedure. Suggested Corrective Action:

Supply a main procedure.

4012 Not enough main storage available. Suggested Corrective Action:

Run the program in a larger region.

4020 Code returned if the program is about to enter a permanent wait state. Suggested Corrective Action:

Check for a WAIT for I/O where no READ or WRITE was requested, or a WAIT for an EVENT not known to another active TASK.

4024 Code returned if a task in a multitasking program has terminated without use of the PL/I termination routines. Suggested Corrective Action:

Check assembler language subroutines.

4028 Excessive fragmentation of storage. PL/I's working storage is in 255 fragments, and more is needed. Suggested Corrective Action:

A larger ISA may help, or change the logic of the program so that files are closed in (more nearly) inverse order to the order in which they were opened and that controlled variables are freed in (more nearly) the inverse of the order in which they were allocated.

4032 The DSA chain fields have been overlaid. Suggested Corrective Action:

Run with the SUBSCRIPTRANGE, STRINGRANGE, and STRINGSIZE conditions enabled. Also, check for logic errors in the use of POINTER variables and CONTROLLED storage.

4036 A program check occurred during an IMS call statement. Register 2 points to the PIE/EPIE. Suggested Corrective Action:

Follow IMS problem determination procedures for program checks.

If a return code in the 4000-4028 range is encountered and the cause cannot be traced to a source program error, it may be necessary to call in IBM program support personnel. Appendix B, "Requirements For Problem Determination And APAR Submission" on page 400, describes the materials that will be required for examination by IBM in such circumstances.

ABEND CODES

Note that 4000, 4024, 4028, 4032 and 4036, included in the above listing of return codes, are also abend codes.

THE ABEND FACILITY

WHEN YOU REALLY NEED AN ABEND

PL/I error handlers attempt to trap most abends and issue a return code instead. Although this works well for batch applications, applications running with a database program such as IMS or CICS often rely on a system-issued abend to initiate a transaction backout. It is possible to get an abend when the ERROR condition is raised by modifying a user exit, IBMBEER, that PL/I supplies.

PL/I ACTION WHEN THE ERROR CONDITION IS RAISED

When the ERROR Condition is raised, PL/I attempts to enter an ERROR on-unit.

The PL/I user exit is taken in these cases: The program terminates with the standard system action and an abend is not issued by the system. This occurs if the ERROR on-unit ends with the END statement or if there is no ERROR on-unit.

The PL/I user exit is not taken in these cases: The program does not terminate with the standard system action and a special return code is issued. This occurs if the ERROR on-unit is terminated by a GOTO, STOP, EXIT, or a call to PLIDUMP with the S or E option. It is assumed that the ERROR on-unit has either corrected the ERROR condition or has itself initiated the transaction backout.

GETTING A SYSTEM-ISSUED ABEND

PL/I supplies the IBMBEER user exit to help with database applications that rely on a system-issued abend.

The IBMBEER exit is entered if an ERROR on-unit ends with the standard system action or if no ERROR on-unit is present. You can replace the IBM-supplied IBMBEER with a user-coded IBMBEER to cause an abend. For an example of a user-coded IBMBEER module see OS PL/I Optimizing Compiler Installation Guide for your system.

The IBMBEER user-coded exit should not contain an ABEND macro. If the exit contains an ABEND macro, PL/I termination will not complete. Storage and other resources will be lost to the database system.

CHAPTER 11. COMMUNICATING BETWEEN PL/I AND ASSEMBLER-LANGUAGE MODULES

OVERVIEW

Writing Assembler language subroutines for PL/I and calling PL/I subroutines from Assembler programs are simple operations, provided that a set of conventions are carefully followed. There are two reasons for the need for these conventions:

1. **PL/I parameter passing conventions:** These are adopted by PL/I to allow the length of nonarithmetic data items to be passed to a called routine.
2. **The PL/I environment:** This is an arrangement of registers and control blocks used by PL/I to simplify error handling, storage management, and other housekeeping tasks.

PARAMETER PASSING

If an Assembler routine is called from PL/I, the parameter problem can be overcome by using the ASSEMBLER option thus:

```
DCL ASMSUB ENTRY OPTIONS(ASSEMBLER),
  CHARSTRING CHAR(25);
CALL ASMSUB(CHARSTRING);
```

This results in the address of the argument being passed directly rather than the address of a control block that contains the length and address of the arguments.

If an Assembler routine is to call PL/I or if PL/I is to use an Assembler routine as a function reference, either the PL/I conventions must be followed or some method must be found of circumventing them. (See also "Arguments, Parameters, Returned Values and Return Codes" on page 308.)

ENVIRONMENT

Assembler subroutines called from PL/I: The PL/I environment causes problems to Assembler subroutines that are called from PL/I mainly because a SPIE/ESPIE macro is used in PL/I to set up an error exit that depends on having register 12 pointing to a PL/I control block known as the TCA (Task Communications Area) and register 13 pointing at a save area following normal save area chaining conventions. When PL/I calls an Assembler subroutine, the subroutine must either forego the use of register 12 or cancel and reissue the SPIE/ESPIE macro instruction, either retaining PL/I error handling or setting up its own. It is normally better to retain PL/I error handling, because the issuing of two SPIE/ESPIE macro instructions is a considerable overhead and PL/I error handling normally gives a useful message when a program check occurs.

For a recursive routine, the PL/I environment provides a ready made LIFO (last-in first-out) storage stack and an overflow mechanism. Assembler routines can use this, but must not use register 12 and should carefully follow the code and instructions in the examples.

PL/I subroutines called from Assembler: When PL/I is called from Assembler, the PL/I environment must be set up before the PL/I subroutine is executed. If the PL/I subroutine is called only once this can be done in the same manner as it is done when a PL/I program is called from the system. To do this the PL/I subroutine should be given the MAIN option and the Assembler subroutine should branch on register 15 to an entry point called

PLICALLA. If the PL/I routine is called a number of times, some device must be employed to prevent the PL/I environment being discarded at the end of each call. This is because setting it up is a significant time overhead. The suggested method is to call a PL/I procedure which has the MAIN option and for this in turn to re-call the Assembler program. In this way the PL/I environment remains available to PL/I subroutines without time overhead. If the Assembler routine was itself called from PL/I, no problems exist because the PL/I environment will already be in existence.

HOW TO WRITE YOUR ROUTINES

Examples in this chapter show the code required to interface between PL/I and Assembler. Provided you bear in mind the notes in the examples, you can use the code as it stands together with your Assembler routines. If you wish to make consistent use of Assembler-PL/I programming you should, however, read the remaining sections of the chapter to understand the reasoning behind the code.

ASSEMBLER ROUTINE CALLED FROM PL/I: Unless you have good reasons for wanting to do your own error handling, you should use the code in Figure 129 on page 301 for non-recursive and non-reentrant routine and the code in Figure 130 on page 302 for a recursive and reentrant routine. If the routine is to receive parameters or to return values, study the section "Arguments, Parameters, Returned Values and Return Codes" on page 308.

ASSEMBLER CALLING PL/I SUBROUTINE: If your PL/I subroutine is invoked only once, it should, if possible, be given the MAIN option and called via entry point PLICALLA thus:

```
L    15,=V(PLICALLA)
BALR 14,15
```

Note that PLICALLA is a standard entry point, not the name of a PL/I routine.

If is impractical to compile the program with the MAIN option, (it might for example, already be compiled as a PL/I subroutine) you can insert its address in PLIMAIN as shown in Figure 126 on page 296 and then call PLICALLA.

If PL/I routines are to be called a number of times you should follow the complete scheme shown in Figure 128 on page 298. If parameters are to be passed or values returned see the "Arguments, Parameters, Returned Values and Return Codes" on page 308.

The remainder of this chapter covers the points summarized above in more detail.

THE PL/I ENVIRONMENT

The PL/I environment is the term used to describe a number of control blocks created by routines that are provided by the OS PL/I Resident and Transient Libraries to satisfy the storage-management and error-handling requirements of a PL/I procedure.

When a PL/I program invokes an Assembler-language routine, the invoked routine must ensure that the PL/I environment is preserved. The PL/I environment is preserved by observing the standard OS/VS linkage conventions, which include the storing of register values in a save area, and by ensuring that the content of register 12 is not modified by the Assembler routine if PL/I is to handle interrupts that occur during execution of the Assembler routine. (It is normally best to allow PL/I to manage error handling. Allowing Assembler language routines to handle their own errors involves a considerable time overhead—the very thing Assembler routines are intended to avoid.) Register 13

must be set to the address of a new save area established by the Assembler routine.

In a mixed environment of PL/I and Assembler-language routines, if SYSPRINT is to be used anywhere, it should be declared and opened in the first PL/I program that gets control. Otherwise, the results are unpredictable.

ESTABLISHING THE PL/I ENVIRONMENT

The PL/I environment is established by the OS PL/I Resident Library routine IBMBPIR and the OS PL/I Transient Library routine IBMBPII for a nonmultitasking program and by IBMTPIR and IBMTPII for a multitasking program. An Assembler-language routine that invokes a PL/I procedure for which the PL/I environment has not been established can use one of three standard entry points to establish the environment. The routine IBMBPIR or IBMTPIR (with IBMBPII or IBMTPII) is entered through a control section which has three standard entry points, PLISTART, PLICALLA, and PLICALLB; which is further described in "Calling PL/I Procedures from Assembler Language" on page 303.

USE OF PLIMAIN TO INVOKE A PL/I PROCEDURE

Once IBMBPIR or IBMTPIR (with IBMBPII or IBMTPII) has created the environment, it transfers control to the PL/I procedure whose address is contained in the compiler-generated control section PLIMAIN. Normally, after link editing, PLIMAIN will contain the entry point address of the first, or only, PL/I main procedure in the program.

PL/I provides three standard entry points for calling PL/I via the initialization routines and PLIMAIN. They differ in the arguments that can be passed, see "Arguments, Parameters, Returned Values and Return Codes" on page 308.

If an Assembler subroutine calls PL/I by one of the standard entry points, control is passed to the initialization routines which, when they have set up the PL/I environment, pass control to the address in PLIMAIN. Thus, if any of the standard entry points are used, the Assembler program must ensure that the address of the required PL/I routine is in PLIMAIN. This will happen if the required procedure has the MAIN option and is the first or only MAIN procedure in the load module. Where this is not the case the address must be set in PLIMAIN as in Figure 126. A PLIMAIN control section will always be available enabling this scheme to be used.

```
        .
        .
        LA    1,ARGLIST         POINT R1 at ARGLIST
        L     2,=V(PLIMAIN)     CHANGE ADDRESS IN PLIMAIN
        L     3,=V(MYPROG)      TO THAT OF
        ST    3,0(2)            MYPROG
        L     15,=V(PLICALLA)   BRANCH TO ADDRESS IN PLIMAIN VIA
        BALR  14,15             PLICALLA WHICH SETS UP PL/I ENVIRONMENT
        .
        .
        .
ARGLIST DC    A(arg1)         FIRST ARGUMENT PASSED TO MYPROG
        DC    A(X'80000000' + arg2)  LAST ARGUMENT PASSED TO MYPROG
        .
        .
```

Figure 126. Inserting a PL/I Entry Point Address in PLIMAIN and Calling the Entry

```
ASM1     CSECT
         DC    C'ASMMAIN',AL1(7)
MAINPROC EQU *                 'MAIN' ENTRY POINT
         STM   14,12,12(13)  SAVE REGS
         BALR  11,0            GET
         USING *,11            ADDRESSABILITY
*                              GET STORAGE FOR A NEW DSA
         LA    0,104           LENGTH REQUIRED 104 BYTES
         L     1,76(13)        ADDRESS OF START OF CURRENTLY-
         ALR   0,1             AVAILABLE STORAGE
         CL    0,12(12)        IS THERE ENOUGH SPACE LEFT?
         BNH   ENOUGH          YES
         L     15,116(12)      LOAD ADDR. OF OVERFLOW ROUTINE
         BALR  14,15           AND BRANCH TO IT.
ENOUGH   EQU   *
         ST    0,76(1)         STORE ADDRESS OF START OF
*                              REMAINING AVAILABLE STORAGE
*                              IN NEW DSA AT OFFSET 76
         ST    13,4(1)         SET BACK CHAIN
         ST    1,8(13)         SET FORWARD CHAIN
         MVC   72(4,1),72(13) COPY ADDRESS OF WORKSPACE FOR
*                              USE BY THE PL/I LIBRARY
         LR    13,1            POINT 13 AT NEW DSA
         MVI   0(13),X'80'     SET FLAGS IN THE DSA TO
         MVI   1(13),X'00'     PRESERVE PL/I
         MVI   86(13),X'91'    ERROR_HANDLING
         MVI   87(13),X'C0'    IN THE ASSEMBLER ROUTINE
         .
         .
   (Your Assembler routine goes here. Register 12 must not be altered.
    PL/I subroutines are called as shown below.)
         .
         .

* CALL   PL/I   SUBROUTINE
         L     15,=V(GETREC)       PL/I INPUT ROUTINE
         BALR  14,15
         .
         .
         .
* TERMINATE
* RETURN TO TERMINATE PL/I ENV
         L     13,4(13)
         LM    14,12,12(13)        RESTORE REGS.
         LR    0,13                GET A(TERMINATING DSA)
         BR    14                  RETURN TO PL/I TERMINATION ROUTINES
         ENTRY PLIMAIN             MAKE NAME 'KNOWN' EXTERNALLY
PLIMAIN  DC    A(MAINPROC)         'MAIN PROCEDURE'
         END
```

Note: This code results in the entire program running in the PL/I environment. It is an alternative to the scheme in Figure 128 on page 298. The load module entry point invoked by the system is PLISTART which passes control via PLIMAIN to MAINPROC.

Figure 127. Skeletal Code for an Assembler Program that Calls PL/I Subroutines a Number of Times

Once the PL/I environment has been established, it can, as shown in Figure 127 and Figure 128 on page 298, be preserved, and any PL/I procedure can be invoked subsequently by loading the address of its entry point into register 15, and executing a branch-and-link-register instruction to it.

```
//OPT13#3  JOB
//STEP1     EXEC ASMHC,PARM.C='OBJECT,NODECK'
//C.SYSLIN DD DSN=&&LOADSET,UNIT=SYSDA,DISP=(NEW,PASS),
//         SPACE=(80,(200,100)),DCB=BLKSIZE=80
//C.SYSGO DD DUMMY
//C.SYSIN DD *
MYPROG   CSECT
         ENTRY ASSEM
         STM   14,12,12(13)          ESTABLISH SUPERVISOR REGISTERS
         BALR  10,0                  ESTABLISH ADDRESSABILITY
         USING *,10
         LA    4,SAVEAREA            CURRENT SAVE AREA ADDRESS
         ST    13,4(4)               STORE CHAINBACK ADDRESS
         ST    4,8(13)               STORE CHAIN FORWARD ADDRESS
         LR    13,4                  STORE CURRENT SAVE AREA ADDRESS
*
         SR    1,1                   SET REGISTER 1 TO ZERO WHEN
*                                    A PARAMETERLESS ENTRY POINT TO
*                                    PROCEDURE THAT DOES NOT RETURN
*                                    A VALUE IS TO BE INVOKED
*
         L     15,=V(PLICALLA)       CALL THE PL/I PROCEDURE THAT
         BALR  14,15                 HAS OPTIONS(MAIN) AND SO SET
*                                    UP THE PL/I ENVIRONMENT AND
*                                    THEN CALL ASSEM.
*
         L     13,4(13)              ON RETURNING FROM PL/I
         L     14,12(13)             RESTORE REGISTERS
         LM    1,12,24(13)           AND
         BR    14                    RETURN TO THE SUPERVISOR.
*
         DC    C'ASSEM'              THE NAME IN PL/I FORMAT
         DC    AL1(5)
ASSEM    DS    0H
         STM   14,12,12(13)          STORE PL/I REGISTERS
         BALR  10,0                  FOR PROCEDURE "MAIN"
         USING *,10                  ESTABLISH ADDRESSABILITY
*                                    GET STORAGE FOR A NEW DSA
         LA    0,104                 LENGTH REQUIRED 104 BYTES
         L     1,76(13)              ADDRESS OF START OF CURRENTLY-
         ALR   0,1                   AVAILABLE STORAGE
         CL    0,12(12)              IS THERE ENOUGH SPACE LEFT?
         BNH   ENOUGH                YES
         L     15,116(12)            LOAD ADDR. OF OVERFLOW ROUTINE
         BALR  14,15                 AND BRANCH TO IT.
ENOUGH   EQU   *
         ST    0,76(1)               STORE ADDRESS OF START OF
*                                    REMAINING AVAILABLE STORAGE
*                                    IN NEW DSA AT OFFSET 76
         ST    13,4(1)               SET BACK CHAIN
         ST    1,8(13)               SET FORWARD CHAIN
         MVC   72(4,1),72(13)        COPY ADDRESS OF WORKSPACE FOR
*                                    USE BY THE PL/I LIBRARY
         LR    13,1                  POINT 13 AT NEW DSA
         MVI   0(13),X'80'           SET FLAGS IN THE DSA TO
         MVI   1(13),X'00'           PRESERVE PL/I
         MVI   86(13),X'91'          ERROR-HANDLING
         MVI   87(13),X'C0'          IN THE ASSEMBLER ROUTINE
*
         SR    5,5                   R5 MUST BE ZERO WHEN CALLING
*                                    AN EXTERNAL PL/I PROCEDURE.
*
         SR    1,1                   SET REGISTER 1 TO ZERO WHEN
*                                    A PARAMETERLESS ENTRY POINT TO
*                                    PROCEDURE THAT DOES NOT RETURN
*                                    A VALUE IS TO BE INVOKED
```

Figure 128 (Part 1 of 2). Invoking PL/I Procedures from an Assembler Routine

```
*
         L     15,=V(HEAD)           CALL PL/I TO 'HEAD' PAGE
         BALR  14,15
*
LOOP     EQU   *
         LA    1,ARGTLST1
         L     15,=V(PLIN)           CALL PL/I TO READ AND ADD
         BALR  14,15
*
         L     3,RESULT
         LTR   3,3                   TEST RESULT AND
         BM    OUTLOOP               BRANCH OUT IF IT IS NEGATIVE.
*
         LA    1,ARGTLST2
         L     15,=V(PLOUT)          CALL PL/I TO OUTPUT RESULT
         BALR  14,15
         B     LOOP
*
OUTLOOP  EQU   *
         SR    1,1                   SET REGISTER 1 TO ZERO
         L     15,=V(FOOT)           CALL PL/I TO 'FOOT' PAGE
         BALR  14,15
*
         L     13,4(13)              RETURN TO THE PL/I PROC WITH
         LM    14,12,12(13)          OPTIONS(MAIN).
         BR    14
*
ARGTLST1 DC    A(DATA)
ARGTLST2 DC    A(X'80000000' + RESULT)
DATA     DC    F'123'
RESULT   DC    F'0'
SAVEAREA DC    18F'0'
         END   MYPROG
/*
//STEP2  EXEC PLIXCLG
//PLI.SYSIN DD *
* PROCESS;
  MAIN: PROC OPTIONS(MAIN);
        DCL ASSEM ENTRY;
        CALL ASSEM;
        END;
* PROCESS;
  PLIN:  PROC(I) RETURNS(FIXED BIN(31));
         DCL (I,J) FIXED BIN(31);
         GET LIST(J);
         RETURN(I+J);
  HEAD:  ENTRY;
         PUT LIST('THE FIRST LINE OF OUTPUT AT THE TOP OF THE PAGE')
             PAGE;
         PUT SKIP(2);
         END;
* PROCESS;
  PLOUT: PROC(K);
         DCL K FIXED BIN(31);
         PUT LIST(K);
         RETURN;
  FOOT:  ENTRY;
         PUT LIST('END OF THE OUTPUT FOR THIS JOB') SKIP(2);
         END;
/*
//GO.SYSIN DD *
 50  77  123  234  345  456  -23  -100  -123  -234
/*
```

Figure 128 (Part 2 of 2). Invoking PL/I Procedures from an Assembler Routine

THE DYNAMIC STORAGE AREA (DSA) AND SAVE AREA

Whenever a PL/I procedure is invoked, it requires for its own use a block of storage known as a dynamic storage area (DSA). A DSA for a PL/I procedure consists of a save area for the contents of registers, a backchain address that points to the save area for the previous routine, and storage for variables and miscellaneous housekeeping items.

An Assembler routine invoked from PL/I should take one of the following actions to allow the PL/I DSA chain to function. The choice depends on whether the Assembler routine needs to set up its own error exit and whether or not it, in turn, calls further PL/I routines.

1. If the Assembler routine is not to set up its own error exit, it must store the contents of all registers in the existing PL/I DSA and establish its own save area in which the backchain address of the PL/I DSA must be stored. The first byte of the save area must be set to zero unless the save area is in PL/I's storage (see "Invoking a Recursive or Reentrant Assembler Routine" on page 301). The second word of the save area is the backchain address. The remainder of the save area would only be used by a routine invoked from the Assembler routine or by the PL/I error handler, if used, for saving the Assembler routine's registers.

2. If the Assembler routine is going to set up its own error exit and does not invoke a further PL/I routine, the SPIE/ESPIE macro must be used to reset the interrupt handler but only those registers that it modifies must be stored. The SPIE macro is also discussed in "Overriding and Restoring PL/I Error-Handling" on page 306.

CALLING ASSEMBLER ROUTINES FROM PL/I

INVOKING A NONRECURSIVE AND NONREENTRANT ASSEMBLER ROUTINE

When a PL/I program invokes a nonrecursive and nonreentrant Assembler-language routine, the Assembler-language routine must follow OS/VS linkage conventions and save the registers for use by PL/I on return from the Assembler-language routine. The register values are stored in the PL/I DSA, the address of which is contained in register 13 on entry to the Assembler-language routine. This address must then be stored in the backchain word in a save area defined by the Assembler routine itself. The appropriate Assembler instructions should be executed immediately when the Assembler routine is invoked in order to achieve the given objectives. Before returning to the PL/I routine, the Assembler routine must restore the registers to the values held when the PL/I routine invoked the Assembler routine.

If register 13 does not point to a valid DSA, PL/I error handling may not work properly while the Assembler routine is executing. If register 13 does not point to a valid DSA, the Assembler routine must replace PL/I error handling as described in "Overriding and Restoring PL/I Error-Handling" on page 306.

Figure 127 on page 297, Figure 128 on page 298, and Figure 130 on page 302, give examples of how to create a valid DSA. For an example of a program that assumes no interrupts, see Figure 129 on page 301. The example in Figure 129 on page 301 also assumes that the Assembler routine uses register 10 as its base register. If you insert your Assembler instructions at the point indicated in the figure your Assembler subroutine can be called by a PL/I CALL statement and you need have no knowledge of the PL/I environment.

To be valid for error handling purposes, the DSA pointed to by register 13 must:

- Be 88 bytes long
- Begin on a double word boundary
- Bytes 0 and 1 must contain x'8000'
- The word at offset 72 must point to an available library work space (the word at offset 72 in the previous DSA does this)
- The word at offset 76 must contain the address of the next available byte of PL/I storage, which must be the address of a double word
- Bytes 86 and 87 must contain x'91C0'
- The PL/I LIFO stack storage management scheme should be used

INVOKING A RECURSIVE OR REENTRANT ASSEMBLER ROUTINE

A recursive or reentrant Assembler routine invoked from PL/I must obtain a separate save area for each invocation, and so cannot use the method of having a static save area illustrated in Figure 129. The suggested method, described in Figure 128 on page 298, is to make use of the PL/I storage management scheme. This obtains storage in a LIFO (last in/first out) stack and can use the PL/I storage overflow routine to attempt to obtain further storage when the storage initially available for dynamic use by the program is used up. This method is referred to as obtaining a DSA.

The first byte of a DSA set up using the PL/I storage scheme is used in PL/I error handling. Consequently, it must be set to a special value depending on whether you wish to use PL/I error handling in the Assembler routine.

```
DUMREC     CSECT
           ENTRY SRCH
           DC    C' SRCH'
           DC    AL1(5)
SRCH       DS    0H
           STM   14,11,12(13)     STORE PL/I REGISTERS IN PL/I DSA
           BALR  10,0             ESTABLISH BASE REGISTER
           USING *,10
           LA    4,SAVEAREA
           ST    13,SAVEAREA+4    STORE PL/I DSA ADDRESS IN SAVE AREA
           ST    4,8(13)
           LA    13,SAVEAREA      LOAD SAVE AREA ADDRESS
           .
           .                      ASSEMBLER
           .                      ROUTINE
           .
           L     13,4(13)         RESTORE PL/I REGISTERS
           LM    14,11,12(13)     AND
           BR    14               RETURN TO PL/I
SAVEAREA   DC    20F'0'           ALLOCATE 80 BYTE SAVE AREA
           .
           .
```

Figure 129. Skeletal Code for a Non-Recursive Assembler Routine to be Invoked from PL/I

The DSA can be used in the PL/I manner to obtain storage that will be unique to each invocation. See Figure 130.

Additional storage can be obtained within the DSA for use by each invocation of the Assembler routine. The total length of the DSA must be a multiple of 8 bytes.

```
DUMREC   CSECT
         ENTRY REC
         DC    C'REC'
         DC    AL1(3)
REC      DS    0H
         STM   14,11,12(13)  STORE CALLER'S REGISTERS IN CALLER'S DSA
         BALR  10,0          ESTABLISH BASE REGISTER
         USING *,10
         LR    4,1           SAVE ANY PARAMETER LIST ADDRESS
*                            PASSED FROM CALLING ROUTINE
         LA    0,96          PUT THE LENGTH OF THE REQUIRED DSA IN REG 0
         L     1,76(13)      LOAD THE ADDRESS OF THE NEXT AVAILABLE
*                            BYTE OF STORAGE AFTER THE CURRENT DSA
         ALR   0,1           ADD ADDRESSES
         CL    0,12(12)      COMPARE RESULT WITH ADDRESS OF LAST AVAILABLE
*                            BYTE IN STORAGE THAT CAN BE USED
         BNH   ENOUGH
         L     15,116(12)    LOAD AND BRANCH TO THE PL/I STORAGE OVERFLOW
         BALR  14,15         ROUTINE TO ATTEMPT TO OBTAIN MORE STORAGE
ENOUGH   EQU   *
         ST    0,76(1)       STORE THE ADDRESS OF THE NEXT AVAILABLE
*                            BYTE IN STORAGE AFTER THE NEW DSA
         ST    13,4(1)       STORE THE CHAIN-BACK ADDRESS OF THE PREVIOUS
*                            DSA IN THE CURRENT DSA
         MVC   72(4,1),72(13) COPY ADDRESS OF LIBRARY
*                            WORKSPACE
         LR    13,1          STORE THE ADDRESS OF THE NEW DSA IN REGISTER 13
         MVI   0(13),X'80'   SET FLAGS IN DSA TO
         MVI   1(13),X'00'   PRESERVE PL/I
         MVI   86(13),X'91'  ERROR-HANDLING
         MVI   87(13),X'C0'  IN THE ASSEMBLER ROUTINE
         -
         -                   ASSEMBLER
         -                   ROUTINE
         -
         L     13,4(13)      RELEASE CURRENT DSA
         LM    14,11,12(13)  RESTORE CALLER'S REGISTERS
         BR    14
         .
         .
```

DSA Length: In addition to its use as a save area, the DSA can be used for working storage that will be unique to each invocation. Such storage starts at offset 88 and must be acquired in multiples of 8 bytes so that total DSA length is a multiple of 8. The routine above uses 8 bytes giving a total DSA length of 96.

Figure 130. Skeletal Code for a Recursive or Reentrant Assembler Routine to be Invoked from PL/I

Also the entry point should be preceded by the name of the Assembler program, so that the name can be printed in error messages and PLIDUMP. This should be aligned so that the character string name immediately precedes a 1-byte length field (containing the length of the name in hex), which immediately precedes the entry point of the Assembler routine.

USE OF REGISTER 12

If an Assembler routine that modifies register 12 is to be invoked by a PL/I procedure, any program-check interrupts will result in an unpredictable program failure unless the routine establishes its own error handling for program-check interrupts. Consequently, the routine should be amended to use a register other than register 12 so that the PL/I error-handler can be used, or it can issue a supervisor SPIE/ESPIE or STAE/ESTAE macro to establish its own program interrupt or abnormal termination handling facilities. The routine must subsequently restore PL/I error-handling facilities before returning to PL/I. This is discussed further in "Overriding and Restoring PL/I Error-Handling" on page 306. (A routine that changes the content of register 12 should also store it on entry and restore it on return.)

CALLING PL/I PROCEDURES FROM ASSEMBLER LANGUAGE

The simplest way to invoke a single external PL/I procedure from an Assembler-language routine is to give the PL/I procedure the MAIN option and invoke it using entry point PLICALLA. All that is required is to load the address of PLICALLA into register 15 and then to branch and link to it. When PLICALLA is used in this way, the PL/I environment is created and control is then passed by way of PLIMAIN to the first (or only) main PL/I procedure in the program. Use of this technique will cause the PL/I environment to be established separately for each invocation. If the routine you require is not the first or only MAIN procedure, you can put its address in PLIMAIN using the code shown in Figure 126 on page 296.

ESTABLISHING THE PL/I ENVIRONMENT FOR MULTIPLE INVOCATIONS

If the Assembler routine is to invoke either a number of PL/I routines or the same PL/I routine repeatedly, the creation of the PL/I environment for each invocation will be unnecessarily inefficient. The solution is to create the PL/I environment once only for use by all invocations of PL/I procedures. This can be achieved by invoking a main PL/I procedure which immediately reinvokes the Assembler routine. The Assembler routine must preserve the PL/I environment and is then able to invoke any number of PL/I procedures directly. The example in Figure 128 on page 298 contains an Assembler-language routine that establishes the PL/I environment once only for multiple invocations of PL/I procedures. An alternative is to establish the Assembler routine as a 'main procedure' as shown in Figure 127 on page 297. The code in this figure, used around your Assembler routine will allow PL/I subroutines to be called with the minimum overhead.

In Figure 128 on page 298, the Assembler routine MYPROG receives control initially from the supervisor, and invokes the PL/I main procedure MAIN using the entry point PLICALLA to the PL/I initialization routine. The PL/I procedure MAIN immediately reinvokes the same Assembler routine at the entry point ASSEM. Note that, in this example, this name must be an odd number of characters to ensure that the next instruction is halfword aligned. At this entry point, the PL/I environment is stored, and a new DSA, 104 bytes in length, is created in a manner similar to that previously given for creating a DSA in a recursive or reentrant Assembler-language routine. If there is insufficient room for the new DSA, the PL/I overflow routine is invoked to attempt to obtain storage for the DSA elsewhere in storage.

The instructions in the Assembler routine following the label ENOUGH through to the instruction that loads the address of the PL/I entry point HEAD are concerned with setting up the DSA so that the correct environment exists when the routine invokes the external PL/I procedures PLIN and PLOUT and the secondary entry points within them. These instructions should always be present

in order to preserve the PL/I environment set up by the main procedure for subsequent use by any Assembler-invoked PL/I procedures.

Note that when an external PL/I procedure is invoked, register 5 must be set to zero, and that a PL/I procedure, such as PLIN in this example, that returns a value will assign the value to the last address in the argument list, ARGTLST1. This address is the address of the Assembler-defined storage for RESULT. The high-order bit of the fullword containing the address of RESULT in ARGTLST1 indicates that it is the last fullword in the argument list.

If an Assembler-language routine invokes a PL/I procedure without passing any parameters to it and without expecting any value to be returned from it, register 1 must be set to zero. In this example, the procedure PLIN contains a RETURN (expression) statement, but when invoked through the parameterless entry point HEAD, does not return a value to the invoking routine. Similarly, the procedure PLOUT contains the parameterless entry point FOOT and does not return a value.

An alternative is to set up and discard the PL/I environment for each call to PL/I by calling PLICALLA, PLICALLB, or PLISTART. If this is necessary and the procedure to be called is a PL/I procedure that is not the first (or only) main procedure in the program, the user must insert in PLIMAIN the address of the appropriate entry point to the required PL/I procedure. The example in Figure 126 on page 296 sets the address in PLIMAIN to that of the external entry name MYPROG.

PL/I CALLING ASSEMBLER CALLING PL/I

The information given in the preceding sections should be sufficient to write programs that include a PL/I procedure that invokes an Assembler-language routine that invokes a further PL/I procedure. Figure 128 on page 298 contains an example of a program that performs this type of processing.

A PL/I procedure can invoke an Assembler routine, which in turn invokes a PL/I procedure, that was passed as a parameter to the Assembler routine. The second PL/I procedure can reference variables within the first PL/I procedure, which is the statically containing block. To accomplish this referencing, the Assembler routine must load register 5 from the second word of the Entry Data Control Block, which was passed as a parameter. Register 5 must be loaded before calling the PL/I routine. Figure 131 on page 305 demonstrates this process; the assignment statement in P2 modifies the same variable X which was declared in P1, the statically encompassing block.

```
//STEP1     EXEC ASMFC,PARM.ASM='OBJECT,NODECK'
//ASM.SYSGO  DD DSN=&&LOADSET,UNIT=SYSSQ,DISP=(NEW,PASS),
//          SPACE=(80,(200,100)),DCB=BLKSIZE=80
//ASM.SYSIN DD *
A         CSECT
          ENTRY REC
          B     8(0,15)        ADDED TO BRANCH AROUND THE DECLARATIONS
          DC    C'REC'
          DC    AL1(3)
REC       DS    0H
          STM   14,11,12(13)   STORE CALLER'S REGISTERS IN CALLER'S DSA
          BALR  10,0           ESTABLISH BASE REGISTER
          USING *,10
          LR    4,1            SAVE ANY PARAMETER LIST ADDRESS
*                              PASSED FROM CALLING ROUTINE
          LA    0,96           PUT THE LENGTH OF THE REQUIRED
*                              DSA IN REG 0
          L     1,76(13)       LOAD THE ADDRESS OF THE NEXT AVAILABLE
*                              BYTE OF STORAGE AFTER THE CURRENT DSA
          ALR   0,1            ADD ADDRESSES
          CL    0,12(12)       COMPARE RESULT WITH ADDRESS OF LAST
*                              AVAILABLE BYTE IN STORAGE
*                              THAT CAN BE USED
          BNH   ENOUGH
          L     15,116(12)     LOAD AND BRANCH TO THE PL/I STORAGE
          BALR  14,15          OVERFLOW ROUTINE TO ATTEMPT TO OBTAIN
*                              MORE STORAGE
ENOUGH    EQU   *
          ST    0,76(1)        STORE THE ADDRESS OF THE NEXT AVAILABLE
*                              BYTE IN STORAGE AFTER THE NEW DSA
          ST    13,4(1)        STORE THE CHAIN-BACK ADDRESS OF THE
*                              PREVIOUS DSA IN THE CURRENT DSA
          MVC   72(4,1),72(13) COPY ADDRESS OF LIBRARY
*                              WORKSPACE
          LR    13,1           STORE THE ADDRESS OF THE NEW DSA IN
*                              REGISTER 13
          MVI   0(13),X'80'    SET FLAGS IN DSA TO
          MVI   1(13),X'00'    PRESERVE PL/I
          MVI   86(13),X'91'   ERROR-HANDLING
          MVI   87(13),X'C0'   IN THE ASSEMBLER ROUTINE
*
          L     4,4(13)        ADDRESS OF CALLER'S DSA
          L     4,24(4)        ADDRESS OF PARMLIST PASSED TO ROUTINE A
          L     4,0(4)         ADDR OF THE PARAMETER, I.E. THE
*                              CONTROL BLOCK
          L     15,0(4)        ADDRESS OF THE ENTRY POINT
          L     5,4(4)         DSA ADDRESS OF THE STATICALLY
*                              ENCOMPASSING BLOCK
          SR    1,1            INDICATE NO PARMLIST
          BALR  14,15          CALL THE ROUTINE
*
          L     13,4(13)       RELEASE CURRENT DSA
          LM    14,11,12(13)   RESTORE CALLER'S REGISTERS
          BR    14
          END
/*
```

Figure 131 (Part 1 of 2). Passing Parameters from PL/I to Assembler to PL/I.

```
//STEP2  EXEC PLIXCLG
//PLI.SYSIN      DD *
* PROCESS;
 P1: PROC OPTIONS(MAIN);
     DCL X BIN FIXED;
     DCL A EXTERNAL ENTRY (ENTRY) OPTIONS(ASSEMBLER INTER);

     X=0;
     PUT FILE(SYSPRINT) SKIP EDIT ('X=',X) (A(2),A);
     CALL A(P2);
    GO TO FINISH;

 P2: PROC;
      X=1;
     END P2;

 FINISH: PUT FILE(SYSPRINT) SKIP EDIT ('X=',X) (A(2),A);
 END P1;
/*
```

Figure 131 (Part 2 of 2). Passing Parameters from PL/I to Assembler to PL/I.

ASSEMBLER CALLING PL/I CALLING ASSEMBLER

The information given in the preceding sections should be sufficient to write programs that include an Assembler-language routine that invokes a PL/I procedure that in turn invokes an Assembler-language routine. Figure 128 on page 298 contains an example of a program that performs this type of processing.

OVERRIDING AND RESTORING PL/I ERROR-HANDLING

An Assembler-language routine invoked from PL/I can override PL/I error-handling by issuing its own SPIE/ESPIE macro to handle program interrupts or STAE/ESTAE macro to handle abnormal terminations. Normally there is little advantage in doing this as the PL/I error handler produces meaningful messages when an interrupt occurs in an Assembler routine, and furthermore, PL/I on-units in the calling program can be used.

Under MVS/XA, PL/I Release 5 uses the ESPIE and ESTAE macros. The SPIE and STAE macros are used by PL/I Release 4 and SPIE and ESTAE by PL/I Release 5 under MVS/SP Release 1.

If the Assembler subroutine needs its own error exit, a SPIE/ESPIE macro must be issued. When the SPIE/ESPIE is issued the address of the PL/I PICA or fake PICA must be saved. A routine that cancels PL/I error-handling must restore the PL/I error-handling facilities before returning to the PL/I program. It does this by issuing either a STAE/ESTAE macro with an operand of zero or an execute form of the SPIE/ESPIE macro restoring the saved PL/I PICA or fake PICA, according to the macros used to cancel the PL/I error-handling. The example in Figure 132 on page 307 shows how these macros are used to cancel and subsequently restore PL/I error-handling. The code can be incorporated into your routines if you want to specify your own error exit.

```
//STEP1     EXEC ASMFC,PARM.ASM='LOAD,NODECK'
//ASM.SYSGO DD DUMMY
//ASM.SYSIN DD *
          PRINT NOGEN
ASSEM1    CSECT
*         .                              ASSEMBLER CODE FOR INITIALIZING PL/I
*         .                              ENVIRONMENT GOES HERE [1]
*         .
          SPACE 3
          L     4,16                     GET CVT ADDRESS
          TM    116(4),X'80'             IS IT AN XA SYSTEM?
          BNZ   DOXACODE
*** NOTE ***                             CMS MUST STILL USE STAE MACRO
          ESTAE  ESTAEXIT,PURGE=NONE,ASYNCH=YES,TERM=NO
*                                        STAE MACROS ARE STACKED
*
          SPIE   SPIEEXIT,((1,13),15)
          ST    1,SPIETOOK               NEED TO REMEMBER TOKEN
          B     MAINLINE
          SPACE
DOXACODE  EQU   *                        FOR XA ERROR HANDLING
          SPLEVEL SET=2                  USE XA MACROS
          ESTAE  ESTAEXIT,PURGE=NONE,ASYNCH=YES,TERM=NO
          ESPIE SET,ESPIEXIT,((1,13),15)
          ST    1,SPIETOOK
          SPLEVEL SET=1                  USE OLD LEVEL MACROS
MAINLINE  EQU   *

*         .
*         .                              THIS IS WHERE THE CODE WILL GO
*         .

STOPRUN   EQU    *
          ESTAE  0                       XA AND NON-XA SAME FOR RESET OPTIONS
          L      1,SPIETOOK              PRE-LOAD TOKEN
          L     4,16                     GET CVT ADDRESS
          TM    116(4),X'80'             IS IT AN XA SYSTEM?
          BO    UNDOXACD
          SPIE   MF=(E,(1))
          B     ALDONE
UNDOXACD  EQU   *
          ESPIE  RESET,(1)               RESET TO PL/I ERROR HANDLER
          ESTAE  0                       POP OFF ESTAE ENVIRONMENT
ALDONE                                   RETURN TO COMPILED CODE
*         .
*         .                              ASSEMBLER CODE FOR TERMINATING PL/I
*         .                              ENVIRONMENT GOES HERE [1]
ESPIEXIT  EQU   *
          BR    14
SPIEEXIT  EQU   *
          BR    14
ESTAEXIT  EQU   *
          SETRP WKAREA=(1),DUMP=NO,RC=4,RETADDR=STOPRUN
          BR    14
SPIETOOK  DC   F'0'                     SAVE AREA FOR SPIE/ESPIE TOKENS
          IHASDWA DSECT=YES,VRAMAP=NO
      END
//
```

[1]For an example of this code, see Figure 127 on page 297.

Figure 132. Method of Overriding and Restoring PL/I Error-Handling

ARGUMENTS, PARAMETERS, RETURNED VALUES AND RETURN CODES

Arguments are passed between PL/I and Assembler routines by means of lists of addresses known as "parameter lists." Each address in a parameter list occupies a fullword in main storage. The last fullword in the list has its high-order bit turned on to enable it to be recognized. If the parameter list is being passed to a function reference, then the last word corresponds to the return value field in the function.

Each address in a parameter list is either the address of a data item or the address of a control block that describes a data item. Data items themselves are never placed directly in parameter lists.

RECEIVING ARGUMENTS IN AN ASSEMBLER-LANGUAGE ROUTINE

When an Assembler routine is invoked by a PL/I routine by means of a CALL statement or a function reference, the Assembler routine will receive the address of a parameter list in register 1. The meaning of the addresses in the parameter list depends upon whether or not the entry point of the Assembler routine has been declared with the ASSEMBLER option. These two cases are discussed separately in the following paragraphs. The ASSEMBLER option is fully described in the language reference manual for this compiler.

ASSEMBLER ROUTINE ENTRY POINT DECLARED WITH THE ASSEMBLER OPTION

The ASSEMBLER option is provided to simplify the passing of arguments from PL/I to Assembler routines. It specifies that the parameter list set up by PL/I is to contain the addresses of actual data items, rather than the addresses of control blocks, irrespective of the types of data that are being passed. Thus if, for example, an array is passed from PL/I to an Assembler routine, the address in the parameter list is that of the first element of the array.

Note that if a particular data item is not byte-aligned (for example, an unaligned bit string), the address in the parameter list is that of the byte that contains the start of the data item. Also, varying length character and bit strings are preceded in storage by a 2-byte field specifying the current length of the string, and it is the address of this prefix that is placed in the parameter list.

An Assembler routine whose entry point has been declared with the ASSEMBLER option can be invoked only by means of a CALL statement. It cannot be used as a function reference.

ASSEMBLER ROUTINE ENTRY POINT DECLARED WITHOUT THE ASSEMBLER OPTION

If the entry point of the Assembler routine has not been declared with the ASSEMBLER option, each address in the parameter list is either the address of a data item or the address of a control block, depending on the type of data that is being passed.

For arithmetic element variables, the address in the parameter list is that of the variable itself. For all other problem data types, the address in the parameter list is that of a control block known as "locator/descriptor." For program control data, the address in the parameter list is that of a control block. The formats of locator/descriptors and of control blocks for program control data are given in OS PL/I Optimizing Compiler: Execution Logic.

It is recommended that the use of this type of linkage is avoided wherever possible. Access to locator descriptors is normally only necessary when the full attributes of the arguments are not known by the Assembler routine. The use of

function references (which cannot be used with the ASSEMBLER option) can be avoided by passing the receiving field as a parameter to the Assembler routine.

PASSING ARGUMENTS FROM AN ASSEMBLER-LANGUAGE ROUTINE

Arguments can be passed from Assembler to PL/I either when the PL/I environment is active, or when it is not. When the environment is not active execution time options can be passed to the PL/I initialization routines, as well as arguments to the PL/I programs. When it is active, arguments can be passed with their addresses in a list addressed from register 1 provided PL/I conventions are followed.

ARGUMENTS FROM ASSEMBLER WHEN PL/I ENVIRONMENT SET UP

In order to pass one or more arguments to a PL/I routine when the PL/I environment is active, an Assembler routine must create a parameter list and set its address in register 1. The last fullword in the parameter list must have its high-order bit turned on. If the PL/I routine executes a RETURN(expression) statement, the last address in the parameter list must be that of the field to which PL/I is to assign the returned value.

Each address in the parameter list must be either the address of a data item or the address of a control block that describes a data item, depending upon the type of data that is being passed. For arithmetic element variables, the address in the parameter list must be that of the variable itself. For all other problem data types, the address in the parameter list must be that of a locator/descriptor. For program control data, the address in the parameter list must be that of a control block. The formats of locator descriptors and of control blocks for program control data are given in the execution logic manual for this compiler. Information on what is passed between routines for arguments of various data types is indexed in OS PL/I Optimizing Compiler: Execution Logic, under "arguments."

In some cases, it is possible to avoid the use of locator/descriptors when passing aggregates or strings by pretending that the data is an arithmetic variable. Suppose, for example, that an Assembler routine is required to pass a fixed-length character string of 20 characters to a PL/I routine. The Assembler routine can place the address of the character string itself in the parameter list, and the PL/I routine can be written thus:

```
PP:PROC(X);
   DCL X FIXED,
       A CHAR(20) BASED(P);
   P = ADDR(X);
   .
   .
   .
```

Because X is declared to be arithmetic, the address in the parameter list is interpreted as the start of the data that is being passed. This address is assigned to P, and is subsequently used as a locator for the based character string A, which has the attributes of the data that has actually been passed.

This technique will work for all data types except unaligned bit strings. Note that the dummy arithmetic parameter need not have the same length as the data that is actually being passed; it is used simply to enable the passed address to be identified as the start of the data.

ARGUMENTS FROM ASSEMBLER WHEN PL/I ENVIRONMENT IS NOT SET UP

When the PL/I environment is **not** set up, one of three standard entry points PLICALLA, PLICALLB, or PLISTART is used to set up the environment and call the PL/I program. These three differ in whether execution time options as well as PL/I arguments can be passed and the conventions used for passing the arguments and options.

In essence, the differences are as follows:

PLISTART
Execution time options can be passed plus 1 character string argument as for a PL/I main procedure.

PLICALLA
No execution time options arguments can be passed. Other arguments can be passed as to a normal PL/I subroutine.

PLICALLB
Execution time options and a number of arguments can be passed but the linkage is more complex.

Details are as follows:

For PLISTART, the Assembler language routine must insert in register 1 the address of a fullword which in turn contains the address of a halfword prefix to a character string. The character string, which must start on a fullword boundary, can contain a parameter string similar to that which can be specified in the PARM field of a JCL EXEC statement; for example, 'ISASIZE(4K),R/INPUT'. The halfword prefix must contain the number of characters in the string excluding the halfword prefix. This entry point is useful when a PL/I routine is "attached" by an Assembler routine, because the entry point of the PL/I routine does not have to be changed. The use of PLISTART is illustrated in Figure 133 and Figure 134 on page 311.

```
         .
         .
         LA    1,PLISTHWD    GET PLIST ADDRESS
         ATTACH  EP=PLIPROG  ATTACH PL/I PROGRAM
*
PLISTHWD DS    0F
         DC    A(X'80000000' + PLISTHW)   FLAG LAST WORD OF PLIST
PLISTHW  DC    AL2(L'PLISTCH) LENGTH OF PARM STRING
PLISTCH  DC    C'ISASIZE(8K),R/INPUT'   PARM DATA
         .
         .
```

Figure 133. Use of PLISTART for ATTACH

```
          .
          .
          LA     1,PLISTHWD          GET PLIST ADDRESS
          L      15,=V(PLISTART)     GET PL/I ENTRY POINT
          BALR   14,15               CALL PL/I ROUTINE
*
*
          DS     0F
PLISTHWD  DC     A(X'80000000' + PLISTHW)   FLAG LAST WORD OF PLIST
PLISTHW   DC     H'0'
PLISTCH   DC     AL2(0)              NULL PARM STRING
          .
          .
```

Figure 134. Use of PLISTART Passing Null Parameter String

For PLICALLA, the Assembler-language routine must insert in register 1 the address of the argument list that contains the addresses of any arguments to be passed to the PL/I procedure. These must follow the rules described in "Arguments from Assembler when PL/I Environment set up" on page 309. An example is in Figure 135. If no arguments are passed, register 1 should be set to 0.

```
          .
          .
          LA     1,ARGLIST
          L      15,=V(PLICALLA)
          BALR   14,15
          .
          .
ARGLIST   DC     A(ARG1)     ADDRESS OF FIRST ARGUMENT PASSED TO PL/I
          DC     A(ARG2)     ADDRESS OF SECOND ARGUMENT PASSED TO PL/I
          .
          .
          DC     A(X'80000000 + argn or return-value)   END OF ARGUMENT LIST FLAG
*                            ADDRESS OF LAST ARGUMENT
*                            OR RETURNED VALUE
          .
```

Figure 135. Use of PLICALLA

```
          .
          .
          LA    1,ALIST              GET MY PARAMETER LIST
          L     15,=V(PLICALLB)      CALL PL/I LOAD MODULE
          BALR  14,15
          .
          .
ALIST     DC    A(ARGLIST)           ADDRESS OF ARGUMENT LIST
          DC    A(LENGTH)            LENGTH OF STORAGE FOR PL/I
          DC    A(ISA)               ADDRESS OF ISA POINTER
          DC    A(0)                 TASK ISA - NOT USED
          DC    A(0)                 NUMBER OF CONCURRENT SUBTASKS - NONE
          DC    A(OPTIONS)           ADDRESS OF OPTIONS WORD
          DC    A(HPSIZE)            ADDRESS OF SIZE OF HEAP
          DC    A(HEAP)              ADDRESS OF HEAP AREA
          DC    A(HEAPINC)           ADDRESS OF HEAP INCREMENT
          DC    A(0)                 ADDRESS OF SUBTASK HEAP INCREMENT
          DC    A(X'80000000'+ISAINC) END OF LIST + ISA INCREMENT
*
LENGTH    DC    A(1024*8)            LENGTH OF ISA (8K)
ISA       DC    A(ISASTOR)           ADDRESS OF ISA
HEAP      DC    256D'0'              ROUTINES HEAP STARTS HERE
HEAPEND   EQU *
HPSIZE    DC    A(HEAPEND-HEAP)      LENGTH OF HEAP STORAGE
HEAPINC   DC    F'8192'              8K HEAP INCREMENTS.
ISAINC    DC    F'4096'              4K ISA INCREMENTS
OPTIONS   DC    AL1(REPORT+STAE,FREEHEAP+BELHEAP,0,0)
*
*               DEFINITIONS OF BITS IN OPTIONS BYTES
REPORT    EQU   X'80' IN FIRST BYTE
NOREPORT  EQU   X'40' IN FIRST BYTE
SPIE      EQU   X'20' IN FIRST BYTE
NOSPIE    EQU   X'10' IN FIRST BYTE
STAE      EQU   X'08' IN FIRST BYTE
NOSTAE    EQU   X'04' IN FIRST BYTE
COUNT     EQU   X'02' IN FIRST BYTE
NOCOUNT   EQU   X'01' IN FIRST BYTE
*
FLOW      EQU   X'80' IN SECOND BYTE
NOFLOW    EQU   X'40' IN SECOND BYTE
KEEPHEAP  EQU   X'20' IN SECOND BYTE    ( HEAP KEEP )
FREEHEAP  EQU   X'10' IN SECOND BYTE    ( HEAP FREE )
ANYHEAP   EQU   X'08' IN SECOND BYTE    ( HEAP ANYWHERE )
BELHEAP   EQU   X'04' IN SECOND BYTE    ( HEAP BELOW )
MKEPHEAP  EQU   X'02' IN SECOND BYTE    ( MINOR TASK HEAP KEEP )
MFREHEAP  EQU   X'01' IN SECOND BYTE    ( MINOR TASK HEAP FREE )
*
MANYHEAP  EQU   X'80' IN THIRD  BYTE    ( MINOR TASK HEAP ANYWHERE )
MBELHEAP  EQU   X'40' IN THIRD  BYTE    ( MINOR TASK HEAP BELOW )
*
*                           ARGUMENTS TO BE PASSED TO PL/I
ARGLIST   DC    A(ARG1LCT)
          DC    A(X'80000000'+REPLCT)
*                           LOCATOR FOR PARM1
ARG1LCT   DC    AL4(ARGDATA)          ADDRESS OF DATA
          DC    H'8'                  MAX LENGTH OF DATA
          DC    X'8000'               VARYING ATTRIBUTE
*                           LOCATOR FOR RETURNED VALUE
REPLCT    DC    AL4(REP)              ADDRESS OF REPLY AREA
          DC    H'8'                  LENGTH OF DATA
          DC    X'0000'               NON-VARYING ATTRIBUTE
ARGDATA   DC    AL2(L'ARG1)
ARG1      DC    C'PARMIN'             INPUT PARAMETER
RETVAL    DC    CL8'PARMOUT'          SPACE FOR RETURNED VALUE
REP       DC    CL8' '
ISASTOR   DC    1024D'0'              ROUTINES ISA STARTS HERE
```

Figure 136. Use of PLICALLB

For PLICALLB, the Assembler-language routine must insert in register 1 the address of an argument list that contains the items shown below. An example is in Figure 136 on page 312.

- The address of the argument list containing addresses of arguments to be passed to the PL/I routine, and optionally,
- The address of the length of storage to be made available to the program in a nonmultitasking program or the major task in a multitasking program. The default for this length is half the available storage for a nonmultitasking program or 8K bytes for the major task in a multitasking program. The length of the initial storage area (ISA) passed must be a multiple of 8 bytes, so that the ISA both starts and ends on a double-word boundary.
- The start address of the initial storage area (ISA) to be used by the PL/I program. This storage must be aligned on a double word. For further information, refer to the discussion of the ISASIZE option under "Execution-Time Options" on page 31. If this argument is not specified, the ISA will be obtained dynamically with a GETMAIN macro instruction.
- The address of the length of storage to be made available to each of the subtasks in a multitasking program. The default for this length is 8K bytes for each subtask. This value is ignored for a nonmultitasking program. The length of the ISA must be multiple of 8 bytes.
- The address of the maximum number of concurrent subtasks that can be attached at any one time. This value is ignored in a nonmultitasking program. The default for this value is 20.
- The address of the options word, in which the execution-time options for a program compiled by the optimizing compiler are specified. These options are: REPORT; STAE; SPIE; COUNT; and FLOW. They are described under "Execution-Time Options" on page 31. The hexadecimal value for each option is given in Figure 136 on page 312.
- The address of the HEAP size. This value is used for a main program in a non-tasking environment; it is also used in a multitasking program as the size of heap for the main task. If this word points to a full word of zeros, then all storage requests will be made from the ISA.
- The address of the area to be be used as a separate HEAP storage area. This area, if supplied, will be used to satisfy programmer allocations for storage of BASED, CONTROLLED, and AREA storage. HEAP should be allocated in doublewords.
- The address of the HEAP increment. This value, when supplied, is used when a storage request cannot be satisfied within the current HEAP allocation. Storage management will use this value in determining how much more system storage to request. The value used will be the larger of the actual storage size requested or the HEAP increment. The HEAP increment will be rounded to a 4K multiple.
- The address of the subtask HEAP increment. This field is like the HEAP increment described above except that it is used only for subtasks. The value is ignored in a non-tasking environment. The subtask HEAP increment will be rounded to a 4K multiple.

- The address of a number to be used for ISA increment. This value will be used when the ISA is full. On an ISA overflow, the larger of ISAINC and the requested amount of storage will be used to request storage for the system. The ISA increment will be rounded to a 4K multiple.

- The address of a number which should be used for subtask ISA overflow conditions. The previously stated rule applies.

The argument list may be variable in length, but a field may not be skipped. The user who does not wish to specify a skipped field should include a fullword of zeros. The normal convention for the end of a parameter list is followed. The last entry should have the high order bit turned on.

The examples in Figure 135 on page 311 and Figure 136 on page 312 show the use of PLICALLA and PLICALLB to invoke the first (or only) main PL/I procedure in the program. The PL/I programs in these cases do not perform multitasking.

RETURN CODES

Return codes can be passed by Assembler subroutines to the PL/I program in register 15. If the Assembler subroutine is declared with OPTIONS(RETCODE), the value passed will be saved by the PL/I program and will be accessed when the built-in function PLIRETV is called. PL/I statements might take the following form:

```
DCL ASMSUB OPTIONS(RETCODE,ASSEMBLER);
CALL ASMSUB;
/*(ASMSUB could set value in register
   15 indicating whether or not
   continuation was worthwhile)*/
IF PLIRETV ¬=0 THEN STOP;
```

If the entry is not declared with OPTIONS(RETCODE), any return code will be ignored.

PL/I routines set a return code in register 15 only when the PL/I environment is destroyed. The return code is the value specified in PLIRETC plus a value generated by the PL/I housekeeping routines if the program terminated because of an error. (For full details of return codes from PL/I, see under the heading "Return Codes" in Chapter 12.) If you wish to pass return code information between subroutines and Assembler language callers and the environment is required for re-use, some mechanism other than PLIRETC should be employed.

CHAPTER 12. THE SORT PROGRAM

The PL/I compilers provide an interface called PLISRT that allows you to make use of the IBM-supplied sort programs. Thus, a ready-made high-performance sort is available in PL/I without the effort of hand coding.

To use the sort program, you must:

1. Include a call to one of the entry points of the sort interface passing it the information on the fields to be sorted, 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 JCL DD statements or by use of the ALLOCATE command on TSO.

When used from PL/I, the sort program will sort records of all normal lengths on a large number of sorting fields. Data of most types can be sorted into ascending or descending order. The source of the data to be sorted may either be a dataset or a PL/I procedure written by the programmer that the sort program will call each time a record is required for the sort. Similarly, the destination of the sort may be a data set or a PL/I procedure that handles the sorted records.

The use of PL/I procedures allows processing to be done before or after the sort itself, thus allowing a complete sorting operation to be totally handled by a call to the sort interface. It is important to understand that the PL/I procedures handling input or output are called from the sort program itself and will effectively become part of it.

THE SORT PROGRAMS AVAILABLE

PL/I can operate with various sort products, such as OS/VS Sort/Merge, its follow on DFSORT, or a program with the same interface. Both OS/VS Sort/Merge and DFSORT, are releases of the same program product: 5740-SM1.

The following material applies to DFSORT. Because you may use programs other than DFSORT, the actual capabilities and restrictions vary. For these capabilities and restrictions, see the DFSORT Application Programming: Guide, or the equivalent publication for your sort product.

When using these publications, you must be aware that the sort program is called from PL/I by a LINK macro instruction and that this imposes some restrictions. Furthermore, the SORT and the RECORD statements are the only two statements that can be passed to the sort program directly from PL/I. However, all valid control statements can be passed to DFSORT using a SORTCNTL data set. By using these control statements, you can increase the flexibility and efficiency of your sort application. For more detail on how to use control statements, see "Chapter 2, 'Program Control Statements'" in the DFSORT Application Programming: Guide. Bearing these points in mind, it is a simple operation to discover the capabilities and restrictions that will apply to your use of sort. The points at which restrictions may apply are given in "What You Need to Know Before Using Sort" on page 318.

BACKGROUND—HOW THE SORT PROGRAM WORKS

If you want to make the best use of the sort program, it is important to 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. This subroutine has four entry points; A, B, C, and D. Each specifies a different source for the unsorted data and destination for the data when it has been sorted. Thus, 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 form sort) is to be placed on another data set. The CALL PLISRT statement must contain an argument list giving the Sort program information about the data set to be sorted, the fields on which it is to be sorted, the amount of space available, the name of a variable into which sort will place a return code indicating the success or failure of the sort, and the name of any output or input handling procedure that may be used.

The sort interface routine builds an argument list for Sort from the information supplied by the PLISRT argument list and the choice of PLISRT entry point. Control is then transferred to the sort program. If an output or input handling routine has been specified, 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 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 137 on page 317 is a simplified flowchart showing this operation.

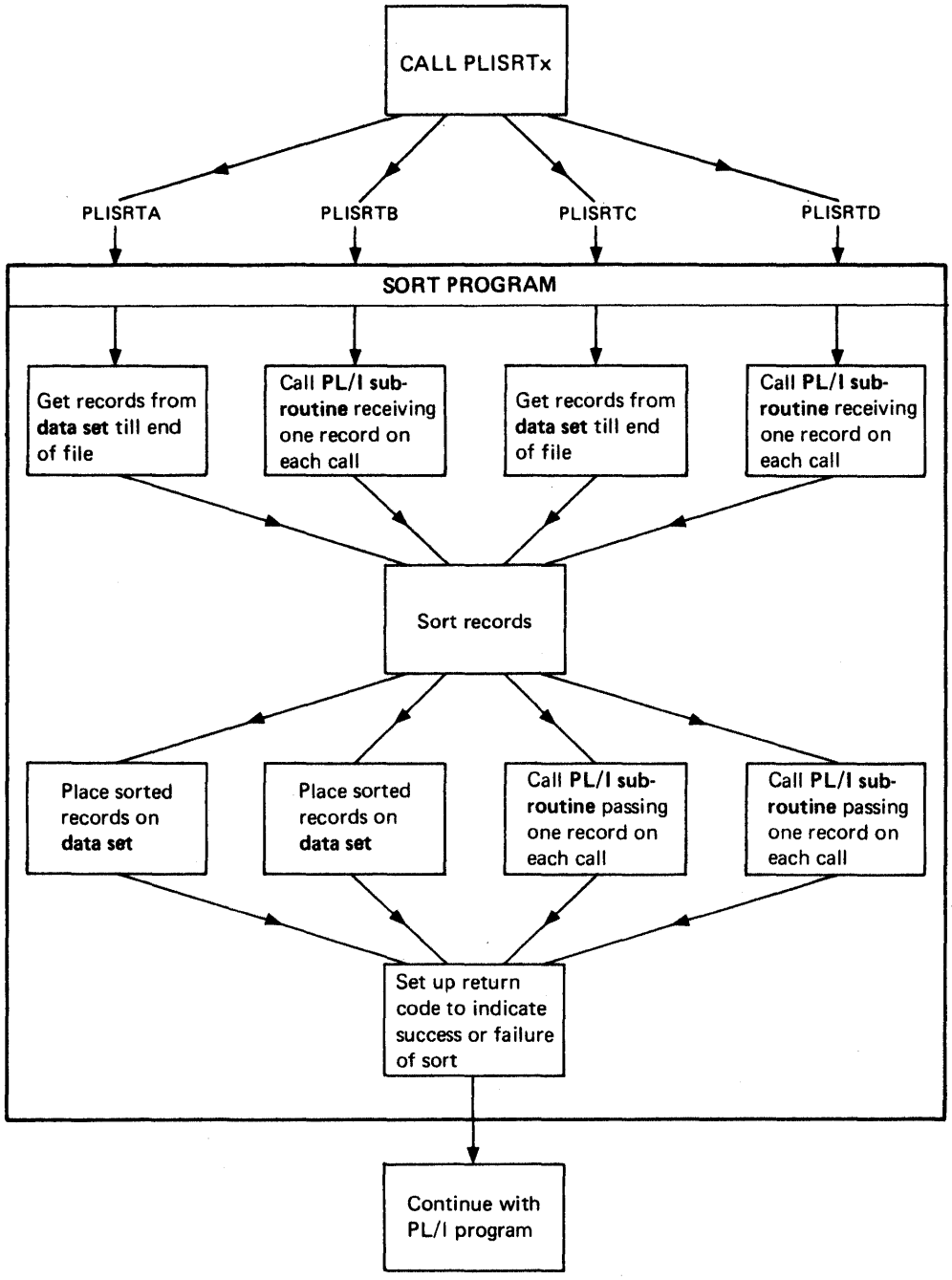


Figure 137. Overview of the Sorting Process

Within the sort program itself, the flow of control between the sort program proper and output- and input-handling routines is controlled by return codes. The sort program calls these routines at the appropriate point in its processing. (Within the sort program, and its associated documentation, these routines are known as user exits. The routine that passes input to be sorted is the E15 exit. The routine that processes sorted input is the E35 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 that it is a self-contained program that handles the complete sort operation, and that 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 PL/I statements required are described and then the data set requirements. The chapter is completed by a series of examples, showing the use of the four entry points of the sort interface routine.

USING THE SORT PROGRAM

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 or in TSO ALLOCATE commands. (The sort interface is not available to PL/I on CMS.)

WHAT YOU NEED TO KNOW BEFORE USING 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 entry point of PLISRT that you will use, you must decide the source of your unsorted data, and the destination of your sorted data. The choice is between data sets and PL/I subroutines. Using data sets is simpler to understand and gives faster performance. Using PL/I subroutines gives you more flexibility and more function, enabling you to manipulate or print the data before it is sorted, and to make immediate use of it in its sorted form. If you decide to use an input or output handling subroutine, you will need to read "Writing the Input and Output Routines" on page 321.

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 a number of things about your data set as follows:

- The position of the sorting fields, these may either be 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 may be altered during sorting.

These are all options of the SORT statement which is the first argument to PLISRT. Having 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 (maximum length for varying).

These are options of the RECORD statement, which is the second argument to PLISRT.

Finally, you must decide on the amount of main and auxiliary storage you must allow for the Sort program. For further details, see "Storage for Sort" on page 328.

THE CALL PLISRT STATEMENT

When you have determined the points described above, you are in a position to write the CALL PLISRT statement. This should be done with some care and the full syntax is given and explained in Figure 141 on page 326 through Figure 143 on page 331. The examples below indicate the form it normally takes.

EXAMPLES OF CALLS TO PLISRT

Example 1

A call to PLISRTA sorting 80-byte records from SORTIN to SORTOUT using 256000 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) ',
           256000,
           RETCODE);
```

Example 2

As above but the input, output, and work data sets are called TASKIN, TASKOUT, and TASKWK01 etc. This might occur if Sort was being called twice in one job step.

```
CALL PLISRTA
    (' SORT FIELDS=(1,80,CH,A) ',
     ' RECORD TYPE=F,LENGTH=(80) ',
     256000,
     RETCODE,
     'TASK');
```

Example 3

As example 1 but the sort is to be undertaken on two fields. First, bytes 1 to 10 which are characters, and then, if these are equal, bytes 11 and 12 which contain a binary field, both fields are to be sorted in ascending order.

```
CALL PLISRTA
     (' SORT FIELDS=(1,10,CH,A,11,2,BI,A) ',
     ' RECORD TYPE=F,LENGTH=(80) ',
     256000,
     RETCODE);
```

Example 4

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) ',
      256000
      RETCODE,
      PUTIN);
```

Example 5

A call to PLISRTD. The input is to be supplied by the PL/I routine PUTIN and the output is to be passed to the PL/I routine PUTOUT. The record to be sorted is 84 bytes varying (including the length prefix). It is to be sorted on bytes 1 through 5 of the data in ascending order, then if these fields are equal, on bytes 6 through 10 in descending order. (Note that the 4-byte length prefix is included so that the actual values used are 5 and 10 for the starting points.) If both these fields are the same, the order of the input is to be retained. (The EQUALS option does this.)

```
CALL PLISRTD
     (' SORT FIELDS=(5,5,CH,A,10,5,CH,D),EQUALS ',
     ' RECORD TYPE=V,LENGTH=(84) ',
     256000,
     RETCODE,
     PUTIN,  /*input routine (sort exit 15)*/
     PUTOUT); /*output routine (sort exit 35)*/
```

TESTING THE RETURN CODE

When the sort completes, Sort sets a return code in the variable named in the fourth argument of the call to PLISRT. It then returns control to the statement that follows the CALL PLISRT statement. The value returned indicates the success or failure of the sort as follows:

```
0   Sort successful
16  Sort failed
20  Sort message data set missing
```

The variable to which the return code is passed must be declared as FIXED BINARY (31,0). It is standard practice to test the value of the return code after the CALL PLISRT 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, the value returned in the sort program should be set as the return code from the PL/I program. This return code is then available for the following job step. See "Execution-time Return Codes" on page 290. The PL/I return code is set by a call to PLIRETC. PLIRETC can be called with the value returned from Sort thus:

```
CALL PLIRETC(RETCODE);
```

This call to PLIRETC should not be confused with the calls made in the input and output routines, where a return code is used for passing control information to Sort.

WRITING THE INPUT AND OUTPUT ROUTINES

The input-handling and output-handling routines are called by Sort when PLISRTB, PLISRTC, or PLISRTD is used. They must be written in PL/I, and can be either internal or external procedures. If they are internal to the routine that calls PLISRT, they will 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.

It should be remembered that the routines will be called individually for each record required by, or passed from Sort. Therefore, each routine must be written to handle one record at a time. Variables declared as AUTOMATIC within the procedures will not retain their values between calls. Consequently, items such as counters, which need to be retained from one call to the next should either be declared as STATIC or be declared in the containing block.

THE INPUT-HANDLING ROUTINE (SORT EXIT E15)

Input routines are normally used to process the data in some way before it is sorted. This may be to print it, as in example Figure 143 on page 331, or may be to generate or manipulate the sorting fields so that the correct results are achieved.

The input handling routine is used by Sort when a call is made to either PLISRTB or PLISRTD. When Sort requires a record, it calls the input routine which should return a record in character string format, and a return code of 12, which means the record passed is to be included in the sort. Sort continues to call the routine until a return code of 8 is passed. This means that all records have already been passed, and 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, V should normally be specified as the record format in the RECORD statement which is the second argument in the call to PLISRT. However, F can be specified, in which case the string will be padded to its maximum length with blanks. The record is returned with a RETURN statement, and the RETURNS attribute must be specified 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 139 on page 323, and skeletal code in Figure 138 on page 322. Examples of an input routine are given in Figure 145 on page 334 and Figure 147 on page 336.

In addition to the return codes of 12 (include current record in sort) and 8 (all records sent), Sort allows the use of a return code of 16. This ends the Sort and sets a return code from Sort to your PL/I program of 16—Sort failed.

It should be noted that a call to PLIRETC sets a return code that will be passed by your PL/I program, and will be available to any job steps that may 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 to avoid receiving a nonzero completion code. By calling PLIRETC with the return code from Sort as the argument, it is possible to make the PL/I return code reflect the success or failure of the sort. This practice is shown in Figure 146 on page 335.

THE 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 in Figure 146 on page 335 and Figure 147 on page 336, or could be 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 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.

```
E15:PROC RETURNS (CHAR(80));
     /*RETURNS attribute must be used specifying length of data to be
        sorted, maximum length if varying strings are passed to Sort.*/
   DCL STRING CHAR(80); /*A character string variable will normally be
                          required to return the data to Sort*/

   IF LAST_RECORD_SENT THEN DO;
      /*A test must be made to see if all the records have been sent,
        if they have, a return code of 8 is set up and control returned
        to Sort*/
        CALL PLIRETC(8);  /*Set return code of 8, meaning last record
                            already sent.*/
        GOTO FINAL;
   END;

   ELSE DO;
   /*If another record is to be sent to Sort, do the
       necessary processing, set a return code of 12
       by calling PLIRETC, and return the data as a
       character string to Sort*/

   ****(The code to do your processing goes here)

       CALL PLIRETC (12);  /*Set return code of 12, meaning this
                             record is to be included in the sort*/
       RETURN (STRING);  /*Return data with RETURN statement*/
   END;
FINAL:
  END;   /*End of the input procedure*/
```

Figure 138. Skeletal Code for an Input Procedure

The record is passed from Sort to the output routine as a character string and a character string parameter must be declared 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. The return code is set 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 the record type was specified as F in the second argument in the call to PLISRT, the parameter should be declared with the length of the record. If the record type was specified as V, the parameter should be declared as adjustable, for example DCL STRING CHAR(*);

Skeletal code for a typical output handling routine is shown in Figure 140, and a flowchart given in Figure 139.

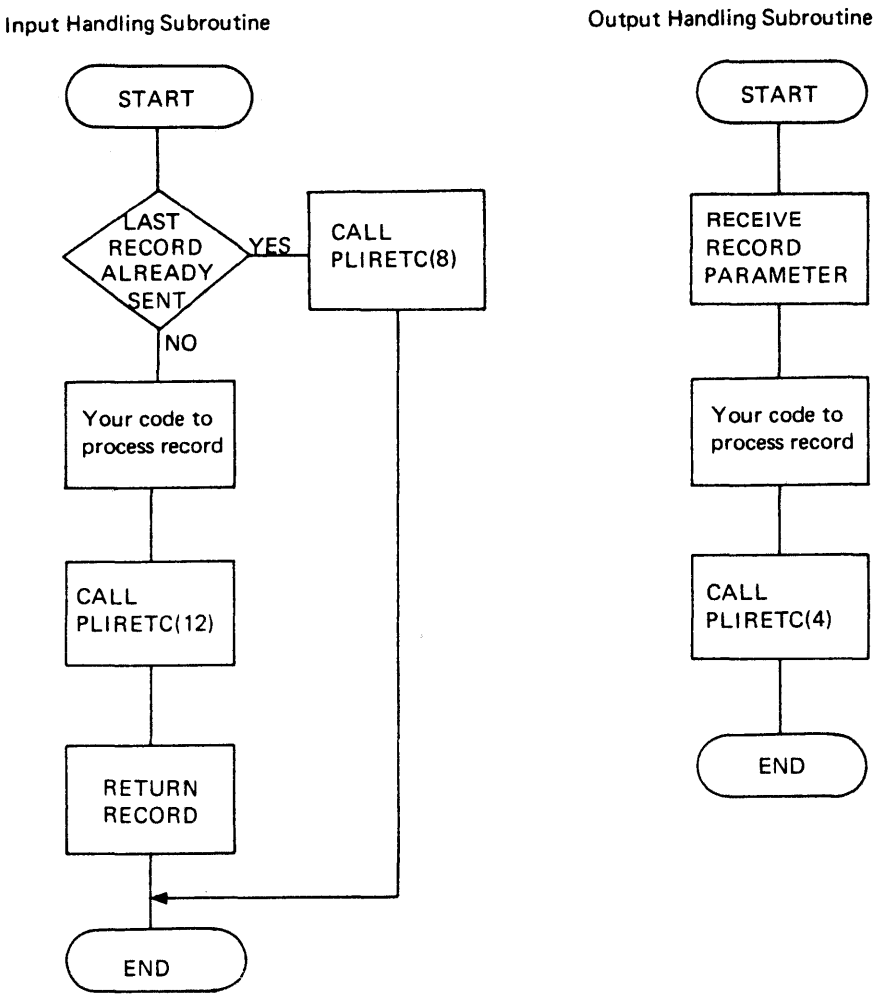


Figure 139. Flowcharts for Input and Output Handling Subroutines

```
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.*/
   ENDVE35;  /*End of procedure returns control to Sort*/
```

Figure 140. Skeletal Code for an Output Handling Procedure

It should be noted 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 may 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 to avoid receiving a nonzero completion code. By calling PLIRETC with the return code from Sort as the argument, it is possible to 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.

DATA SETS FOR SORT

When you call Sort from your PL/I program, it is necessary to specify in your JCL, or through ALLOCATE commands, the data sets required by your Sort. Like PL/I library routines, Sort is a member of the SYS1.LINKLIB or a private library. These Sort data sets must not be open when Sort is called.

These are:

SORTLIB
This library is only required if your working data sets (see below) are on magnetic tape. You must discover the name of this data set from your system programmer.

SYSOUT
A data set (normally the printer) on which messages from the Sort program will be written.

Sort Work data sets

SORTWK01-SORTWK16

Note: If more than 16 are specified, only the first 16 will be used by DFSORT.

xxxxWK01-xxxxWK16
From 1 to 16 working data sets used in the sorting process. These may be direct-access or on magnetic tape. The numbers chosen must be sequential starting from 01. For a discussion of space required and number of data sets, see "Storage for Sort" on page 328.

xxxx represents the 4 characters that can be specified as the data set prefix argument in calls to PLISRT, and allows data sets other than SORTWK to be used. They must start with an alphabetic character and must not be the names PEER, BALN, CRCX, OSCL, POLY, LIST, or DIAG.

Input data set

SORTIN

xxxxIN
The input data set used when PLISRTA and PLISRTC are called.

xxxx represents the 4 characters that can be specified as the data set prefix argument to PLISRT and allow input data sets other than SORTIN to be used. See fuller description under SORTWK above.

Output data set

SORTOUT

******OUT**
The output data set used when PLISRTA and PLISRTB are called.

**** represents the 4 characters that can be specified as the data set prefix argument to PLISRT and allows output data sets other than SORTOUT to be used. See fuller description under SORTWK above.

Checkpoint data set

SORTCKPT

******CKPT**
Data set used to hold checkpoint data, if CKPT or CHKPT option was used in the SORT statement argument. See the DFSORT Application Programming: Guide, or the OS/VS Sort/Merge Programmer's Guide, for information on this program DD statement.

**** See the description under SORTWK.

SORTCNTL

******CNTL**
Dataset from which additional or changed control statements can be read (optional). For additional information on this program DD statement, see DFSORT Application Programming: Guide, or the OS/VS Sort/Merge Programmer's Guide.

**** See the description under SORTWK.

Entry Point	Arguments
PLISRTA Sort input: dataset Sort output: dataset	(sort statement,record statement,storage,return code,[dataset prefix, message level, sort tecnique])
PLISRTB Sort input: PL/I subroutine Sort output: dataset	(sort statement,record statement,storage,return code,input routine, [dataset prefix,message level,sort technique])
PLISRTC Sort input: dataset Sort output: PL/I subroutine	(sort statement,record statement,storage,return code,output routine, [dataset prefix,message level,sort technique])
PLISRTD Sort input: PL/I subroutine Sort output: PL/I subroutine	(sort statement,record statement,storage,return code,input routine,output routine,[dataset prefix,message level,sort technique])
Sort statement	Character string expression containing the sort program SORT statement. Describes sorting fields and format. See Figure 142 on page 329.
Record statement	Character string expression containing the sort program RECORD statement. Describes the length and record format of data. See Figure 143 on page 331.
Storage	Fixed binary expression giving amount of main storage to be made available to the sort program. Must be >54K bytes for OS/VS Sort/Merge and >88K bytes for DFSORT. See also "Storage for Sort" on page 328.
Return code	Fixed binary variable of precision (31,0) in which sort will place a return code when it has completed. The meaning of the return code is as follows: 0=Sort successful 16=Sort failed 20=Sort message data set missing
Input routine	(PLISRTB and PLISRTD only.) Name of the PL/I external or internal procedure that will be 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 that will be passed the sorted records by the sort program from sort exit 35.

Figure 141 (Part 1 of 2). The Entry Points and Arguments to PLISRT

Entry Points	Arguments
Dataset prefix	Character string expression of four characters that will replace the default prefix of 'SORT' in the names of the sort datasets SORTIN,SORTOUT,SORTWK01-SORTWKnn, SORTCNTL, and SORTCKPT if used. Thus if the argument was 'TASK', the datasets TASKIN, TASKOUT, TASKWK01-TASKWKnn, TASKCNTL and TASKCKPT could be used. This facility enables multiple invocations of sort to be made in the same job step. The four characters must start with an alphabetic character and must not be the reserved names PEER, BALN, CRCX, OSCL, POLY, DIAG and LIST. A null string must be coded for this argument if either of the following arguments is required but this is not.
Message level	Character string expression of 2 characters indicating how Sort's diagnostic messages are to be handled as follows: NO No messages to SYSOUT AP All messages to SYSOUT CP Critical messages to SYSOUT SYSOUT will normally be allocated to the printer, hence the use of the mnemonic letter 'P'. Other codes are also allowed for certain of the sort programs. For further details on these codes, see the OS/VS Sort/Merge Programmer's Guide. A null string must be coded for this argument if the following argument is required and this argument is not required.
Sort technique	(This is not used by DFSORT; it appears for compatibility reasons only.) Character string of length 4 that indicates the type of sort to be carried out as follows: PEER Peerage sort BALN Balanced CRCX Criss-cross sort OSCL Oscillating POLY Polyphase sort Normally the sort program will analyze the amount of space available and choose the most effective technique without any action from you. This argument should only be used as a bypass for sorting problems or when you are certain that performance could be improved by another technique. See Sort Programmer's guides for further information.

Figure 141 (Part 2 of 2). The Entry Points and Arguments to PLISRT

STORAGE FOR SORT

Main Storage

As indicated earlier, Sort requires both main and auxiliary storage. The minimum main storage for DFSORT is 88K bytes, but for best performance, more storage (on the order of 1 megabyte) is recommended. You can specify that Sort use the maximum amount of storage available by passing a storage parameter in the following manner:

```
DCL MAXSTOR FIXED BINARY (31,0);
UNSPEC(MAXSTOR)='00000000'B||UNSPEC('MAX');
CALL PLISRTA
    ('SORT FIELDS=(1,80,CH,A) ',
     'RECORD TYPE=F,LENGTH=(80) ',
       MAXSTOR,
       RETCODE,
       'TASK');
```

If files are opened in E15 or E35 exit routines, enough residual storage should be allowed for the files to open successfully.

Auxiliary Storage

Because the minimum auxiliary storage for a particular sorting operation is a complex business, to achieve maximum efficiency, use direct access storage devices (DASDs) when possible, and read the the sections on "Improving Program Efficiency" in the DFSORT Application Programming: Guide or in the OS/VS Sort/Merge Programmer's Guide.

If you are interested only in providing enough storage to ensure the sort will work, make the total size of the SORTWK data sets large enough to hold three sets of the records being sorted. (There is no advantage in specifying more than three if sufficient space can be obtained on three data sets.)

It should be stressed that this is an approximation and will normally result in wasted space. In addition, you cannot be certain that it will always work, so recourse to the sort manuals is advised.

THE SORT STATEMENT - The first Argument to PLISRT

Syntax:

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) '
```

where:

b represents one or more blanks. Blanks shown are mandatory. No other blanks are allowed.

start,length,form,seq define the sorting fields. Any number of such fields can be specified. However 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 the EQUALS option is used. (See later in this figure.) Fields may overlay each other. The maximum total length of the sorting fields is restricted. The current allowed lengths are 256 bytes for all except OS/VS Sort Merge (5740-SM1) where 4092 bytes are allowed. All sorting fields must be within 256 bytes (4092 for OS/VS Sort Merge) of the start of the record.

- **start** is the starting position within the record. The value is given in bytes except for binary data where it is given in a "byte.bit" notation. The first byte in a string is considered to be byte 1, the first bit bit 0. (Thus the second bit in byte 2 is referred to as 2.1.) For varying length records the 4-byte length prefix must be included, making 5 the first byte of data.
- **length** is the length of the sorting field. This is given in bytes except for binary where a "byte.bit" notation can be used. The length of sorting fields is restricted according to their data type, see below.
- **form** is the format of the data. This is the format assumed for the purpose of sorting. All data passed between PL/I routines and Sort must be in the form of character strings. The main data types and the restrictions on their length are shown below. Additional data types are available for special purpose sorts, see your Sort Programmer's Guide.

Figure 142 (Part 1 of 2). The SORT Statement, the First Argument to PLISRT

THE SORT STATEMENT - The first Argument to PLISRT (cont.)

Code	Format	Length
CH	character	1-256 (1-4096 OS/VS Sort Merge)
ZD	zoned decimal signed	1-32
PD	packed decimal signed	1-32
FI	fixed point, signed	1-256
BI	binary, unsigned	1 bit to 256 bytes (4092 OS/VS Sort Merge)
FL	floating-point, signed	1-256

The sum of the lengths of all fields must not exceed 256 bytes (4092 for OS/VS Sort Merge).

seq is the sequence in which the data will be sorted as follows:

A	ascending	that is: 1,2,3 etc.
D	descending	that is: 3,2,1 etc.

Note that E cannot be specified as PL/I does not provide a method of passing a user supplied sequence.

other options
A number of other options can be specified depending on your sort program. These must be separated from the FIELDS operand and from each other by commas. Blanks are not allowed between operands.

FILSZ=y specifies the number of records in the sort and so allows for optimization by Sort. If y is only approximate, it should be preceded by E.

SKIPREC=y specifies that y records at the start of the input file are to be ignored, before sorting the remaining records.

CKPT or CHKPT
specifies that checkpoints are to be taken. If this option is used a SORTCKPT dataset must be provided.

EQUALS
NOEQUALS specifies whether the order of equal records will be preserved as it was in the input (EQUALS) or will be arbitrary (NOEQUALS). Use of the NOEQUALS option may improve Sort performance. The default option is chosen when Sort is installed. The IBM recommended default is NOEQUALS.

DYNALLOC=(d,n)
(OS/VS Sort only) specifies that the program dynamically allocates intermediate storage.

d is the device type (3330, 2314) etc.
n is the number of work areas

Figure 142 (Part 2 of 2). The SORT Statement, the First Argument to PLISRT

```
RECORD STATEMENT - The second argument to PLISRT

Syntax:
     The RECORD statement must be a character string expression which,
     when evaluated, takes the syntax shown below:

     'bRECORDbTYPE=rectype[,LENGTH=(11,[,,14,15])]b'

For example:

    ' RECORD TYPE=F,LENGTH=(80) '

where:

  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
         sorted and unsorted data, the record type must be specified
         as it applies to the work datasets used by Sort.

         If varying length strings are passed to Sort from an input
         routine (E15 exit), V should normally be specified as
         record format, however if F is specified, the records are
         padded to the maximum length with blanks.

         LENGTH specifies the length of the record to be sorted.
         LENGTH can be omitted if PLISRTA or PLISRTC is used,
         because the length will be taken from the input dataset.
         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, the four byte prefix must be
         included.

             11      is the length of the record to be sorted. For
                     VSAM datasets sorted as varying records it is
                     the maximum record size+4.

             ,,      represent two arguments that are not applicable
                     to Sort when called from PL/I. The commas must
                     be included if the arguments that follow are
                     used.

             14      specifies the minimum length of record when
                     varying length records are used. If supplied,
                     it is used by Sort for optimization purposes.

             15      specifies the modal (most common) length of
                     record when varying length records are used. If
                     supplied, it is used by Sort for optimization
                     purposes.
```

Figure 143 (Part 1 of 2). The RECORD STATEMENT—The Second Argument to Sort

RECORD STATEMENT - The second argument to PLISRT (cont.)

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 32756 bytes, and for fixed length records it is 32760 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 may be sorted using a given amount of storage. The minimum block length for tape work units is 18 bytes; the minimum record length is 14 bytes.

Note that the actual maximum depends on storage availability and the track length of the device. See "Calculating Storage Requirements" in the DFSORT Release 6.0 Application Programming: Guide.

Figure 143 (Part 2 of 2). The RECORD STATEMENT—The Second Argument to Sort

```
//OPT14#7  JOB ...
//STEP1 EXEC PLIXCLG
//PLI.SYSIN DD *
 EX106: PROC OPTIONS(MAIN);

     DCL RETURN_CODE FIXED BIN(31,0);

     CALL PLISRTA (' SORT FIELDS=(7,74,CH,A) ',
                   ' RECORD TYPE=F,LENGTH=(80) ',
                     256000,
                     RETURN_CODE);
     SELECT (RETURN_CODE);
       WHEN(0) PUT SKIP EDIT
           ('SORT COMPLETE RETURN_CODE 0') (A);
       WHEN(16) PUT SKIP EDIT
           ('SORT FAILED, RETURN_CODE 16') (A);
       WHEN(20) PUT SKIP EDIT
           ('SORT MESSAGE DATASET MISSING ') (A);
       OTHER    PUT SKIP EDIT (
            'INVALID SORT RETURN_CODE = ', RETURN_CODE) (A,F(2));
       END /* select */;
       CALL PLIRETC(RETURN_CODE);
      /*set PL/I return code to reflect success of sort*/
       END EX106;
//GO.SORTIN DD *
003329HOOKER S.W. RIVERDALE, SATCHWELL LANE, BACONSFIELD
002886BOOKER R.R. ROTORUA, LINKEDGE LANE, TOBLEY
003077ROOKER & SON, LITTLETON NURSERIES, SHOLTSPAR
059334HOOK E.H. 109 ELMTREE ROAD, GANNET PARK, NORTHAMPTON
073872HOME TAVERN, WESTLEIGH
000931FOREST, IVER, BUCKS
/*
//GO.SYSPRINT DD SYSOUT=A,DCB=(RECFM=F,BLKSIZE=80)
//GO.SORTOUT DD SYSOUT=A,DCB=(RECFM=F,BLKSIZE=80)
//GO.SYSOUT DD SYSOUT=A
//GO.SORTWK01 DD UNIT=SYSDA,SPACE=(CYL,2)
/*
```

Figure 144. Example of Sorting from Data Set to Data Set (PLISRTA)

```
//OPT14#8  JOB ...
//STEP1 EXEC PLIXCLG
//PLI.SYSIN DD *
 EX107:  PROC OPTIONS(MAIN);

     DCL RETURN_CODE FIXED BIN(31,0);

     CALL PLISRTB (' SORT FIELDS=(7,74,CH,A) ',
                   ' RECORD TYPE=F,LENGTH=(80) ',
                   256000,
                   RETURN_CODE,
                   E15X);
     SELECT(RETURN_CODE);
       WHEN(0)  PUT SKIP EDIT
           ('SORT COMPLETE RETURN_CODE 0') (A);
       WHEN(16) PUT SKIP EDIT
           ('SORT FAILED, RETURN_CODE 16') (A);
       WHEN(20) PUT SKIP EDIT
           ('SORT MESSAGE DATASET MISSING ') (A);
       OTHER   PUT SKIP EDIT
           ('INVALID RETURN_CODE = ',RETURN_CODE)(A,F(2));
     END /* select */;
     CALL PLIRETC(RETURN_CODE);
      /*set PL/I return code to reflect success of sort*/

 E15X:   /* INPUT HANDLING ROUTINE GETS RECORDS FROM THE INPUT STREAM
            AND PUTS THEM BEFORE THEY ARE SORTED*/
     PROC RETURNS (CHAR(80));
          DCL SYSIN FILE RECORD INPUT,
              INFIELD CHAR(80);

          ON ENDFILE(SYSIN) BEGIN;
             PUT SKIP(3) EDIT ('END OF SORT PROGRAM INPUT')(A);
             CALL PLIRETC(8);  /* signal that last record has
                                  already been sent to sort*/
             GOTO ENDE15;
             END;

           READ FILE (SYSIN) INTO (INFIELD);
           PUT SKIP EDIT (INFIELD)(A(80)); /*PRINT INPUT*/
           CALL PLIRETC(12);  /* request sort to include current
                                 record and refutn for more*/
          RETURN(INFIELD);
    ENDE15:
          END E15X;
  END EX107;
/*
//GO.SYSIN DD *
003329HOOKER S.W. RIVERDALE, SATCHWELL LANE, BACONSFIELD
002886BOOKER R.R. ROTORUA, LINKEDGE LANE, TOBLEY
003077ROOKER & SON, LITTLETON NURSERIES, SHOLTSPAR
059334HOOK E.H. 109 ELMTREE ROAD, GANNET PARK, NORTHAMPTON
073872HOME TAVERN, WESTLEIGH
000931FOREST, IVER, BUCKS
/*
//GO.SYSPRINT DD SYSOUT=A,DCB=(RECMF=F,BLKSIZE=80)
//GO.SORTOUT  DD SYSOUT=A,DCB=(RECMF=F,BLKSIZE=80)
//GO.SYSOUT DD SYSOUT=A
/*
//GO.SORTCNTL DD *
   OPTION DYNALLOC=(3380,2),SKIPREC=2
/*
```

Figure 145. Example of Sorting from Input Handling Routine to Dataset (PLISRTB)

```
//OPT14#9  JOB ...
//STEP1 EXEC PLIXCLG
//PLI.SYSIN DD *
 EX108:  PROC OPTIONS(MAIN);

     DCL RETURN_CODE FIXED BIN(31,0);

     CALL PLISRTC (' SORT FIELDS=(7,74,CH,A) ',
                   ' RECORD TYPE=F,LENGTH=(80) ',
                   256000,
                   RETURN_CODE,
                   E35X);
        SELECT(RETURN_CODE);
          WHEN(0)  PUT SKIP EDIT
              ('SORT COMPLETE RETURN_CODE 0') (A);
          WHEN(16) PUT SKIP EDIT
              ('SORT FAILED, RETURN_CODE 16') (A);
          WHEN(20) PUT SKIP EDIT
              ('SORT MESSAGE DATASET MISSING ') (A);
          OTHER    PUT SKIP EDIT
              ('INVALID RETURN_CODE = ', RETURN_CODE) (A,F(2));
        END /* select */;
     CALL PRLRETC (RETURN_CODE);
      /*set PL/I return code to reflect success of sort*/

 E35X:   /* output handling routine prints sorted records*/
       PROC (INREC);
          DCL INREC CHAR(80);
          PUT SKIP EDIT (INREC) (A);
          CALL PLIRETC(4); /*request next record from sort*/
       END E35X;
    END EX108;
/*
//GO.STEPLIB DD DSN=SYS1.SORTLINK,DISP=SHR
//GO.SYSPRINT DD SYSOUT=A
//GO.SYSOUT   DD SYSOUT=A
//GO.SORTIN DD *
003329HOOKER S.W. RIVERDALE, SATCHWELL LANE, BACONSFIELD
002886BOOKER R.R. ROTORUA, LINKEDGE LANE, TOBLEY
003077ROOKER & SON, LITTLETON NURSERIES, SHOLTSPAR
059334HOOK E.H. 109 ELMTREE ROAD, GANNET PARK, NORTHAMPTON
073872HOME TAVERN, WESTLEIGH
000931FOREST, IVER, BUCKS
/*
//GO.SORTCNTL DD *
   OPTION DYNALLOC=(3380,2),SKIPREC=2
/*
```

Figure 146. Example of Sorting from Data Set to Output Handling Routine (PLISRTC)

```
//OPT14#10  JOB ...
//STEP1 EXEC PLIXCLG
//PLI.SYSIN DD *
 EX109:  PROC OPTIONS(MAIN);
     DCL RETURN_CODE FIXED BIN(31,0);
     CALL PLISRTD ('SORT FIELDS=(7,74,CH,A) ',
                   ' RECORD TYPE=F,LENGTH=(80) ',
                   256000,
                   RETURN_CODE,
                   E15X,
                   E35X);

     SELECT(RETURN_CODE);
       WHEN(0)  PUT SKIP EDIT
           ('SORT COMPLETE RETURN_CODE 0') (A);
       WHEN(20) PUT SKIP EDIT
           ('SORT MESSAGE DATASET MISSING ') (A);
       OTHER    PUT SKIP EDIT
           ('INVALID RETURN_CODE = ', RETURN_CODE) (A,F(2));
       END /* select */;

     CALL PLIRETC(RETURN_CODE);
      /*set PL/I return code to reflect success of sort*/

 E15X:   /* Input handling routine prints input before sorting*/
       PROC RETURNS(CHAR(80));
            DCL INFIELD CHAR(80);

            ON ENDFILE(SYSIN) BEGIN;
               PUT SKIP(3) EDIT ('END OF SORT PROGRAM INPUT.  ',
                   'SORTED OUTPUT SHOULD FOLLOW')(A);
               CALL PLIRETC(8);  /* Signal end of input to sort*/
               GOTO ENDE15;
            END;

            GET FILE (SYSIN) EDIT (INFIELD) (A(80));
            PUT SKIP EDIT (INFIELD)(A);
            CALL PLIRETC(12);  /*Input to sort continues*/
           RETURN(INFIELD);
  ENDE15:
            END E15X;

  E35X:   /* Output handling routine prints the sorted records*/
       PROC (INREC);

            DCL INREC CHAR(80);
            PUT SKIP EDIT (INREC) (A);
     NEXT:  CALL PLIRETC(4); /* Request next record from sort*/
            END E35X;
 END EX109;
/*
//GO.SYSOUT DD SYSOUT=A
//GO.SYSPRINT DD SYSOUT=A
//GO.SORTWK01 DD UNIT=SYSDA,SPACE=(CYL,1)
//GO.SORTWK02 DD UNIT=SYSDA,SPACE=(CYL,1)
//GO.SORTWK03 DD UNIT=SYSDA,SPACE=(CYL,1)
//GO.SYSIN DD *
003329HOOKER S.W. RIVERDALE, SATCHWELL LANE, BACONSFIELD
002886BOOKER R.R. ROTORUA, LINKEDGE LANE, TOBLEY
003077ROOKER & SON, LITTLETON NURSERIES, SHOLTSPAR
059334HOOK E.H. 109 ELMTREE ROAD, GANNET PARK, NORTHAMPTON
073872HOME TAVERN, WESTLEIGH
000931FOREST, IVER, BUCKS
/*
```

Figure 147. Sorting from Input Handling Routine to Output Handling Routine (PLISRTD)

```
//OPT14#11 JOB ...
//STEP1 EXEC PLIXCLG
//PLI.SYSIN DD *
  /* PL/I EXAMPLE USING PLISRTD TO SORT VARIABLE-LENGTH RECORDS */

  EX1306:  PROC OPTIONS(MAIN);
           DCL RETURN_CODE FIXED BIN(31,0);
           CALL PLISRTD (' SORT FIELDS=(11,14,CH,A) ',
                         ' RECORD TYPE=V,LENGTH=(84,,,24,44) ',
                         /*NOTE THAT LENGTH IS MAX AND INCLUDES 4 BYTE
                            LENGTH PREFIX*/
                         256000,
                         RETURN_CODE,
                         PUTIN,
                         PUTOUT);

                SELECT(RETURN_CODE);
                  WHEN(0)  PUT SKIP EDIT (
                           'SORT COMPLETE RETURN_CODE 0') (A);
                  WHEN(16) PUT SKIP EDIT (
                           'SORT FAILED, RETURN_CODE 16') (A);
                  WHEN(20) PUT SKIP EDIT (
                           'SORT MESSAGE DATASET MISSING ') (A);
                  OTHER    PUT SKIP EDIT (
                       'INVALID RETURN_CODE = ', RETURN_CODE) (A,F(2));
                END /* SELECT */;

     CALL PLIRETC(RETURN_CODE);
      /*SET PL/I RETURN CODE TO REFLECT SUCCESS OF SORT*/
      PUTIN: PROC RETURNS (CHAR(80) VARYING);
        /*OUTPUT HANDLING ROUTINE*/
        /*NOTE THAT VARYING MUST BE USED ON RETURNS ATTRIBUTE WHEN USING
          VARYING LENGTH RECORDS*/
          DCL STRING CHAR(80) VAR;

          ON ENDFILE(SYSIN) BEGIN;
               PUT SKIP EDIT ('END OF INPUT')(A);
               CALL PLIRETC(8);
               GOTO ENDPUT;
               END;

          GET EDIT(STRING)(A(80));
          I=INDEX(STRING||'  ','  ')-1; /*RESET LENGTH OF THE */
          STRING = SUBSTR(STRING,1,I); /* STRING FROM 80 TO LENGTH*/
                                       /* OF TEXT IN EACH INPUT   */
                                       /* RECORD.                 */

          PUT SKIP EDIT(I,STRING) (F(2),X(3),A);
          CALL PLIRETC(12);
          RETURN(STRING);
 ENDPUT:  END;
 PUTOUT:PROC(STRING);
         /*OUTPUT HANDLING ROUTINE OUTPUT SORTED RECORDS*/
          DCL STRING CHAR (*);
           /*NOTE THAT FOR VARYING RECORDS THE STRING
             PARAMETER FOR THE INPUT 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;
/*
```

Figure 148 (Part 1 of 2). Example of Sorting Varying Length Records Using Input and Output Handling Routines

```
//GO.SYSIN DD *
003329HOOKER S.W. RIVERDALE, SATCHWELL LANE, BACONSFIELD
002886BOOKER R.R. ROTORUA, LINKEDGE LANE, TOBLEY
003077ROOKER & SON, LITTLETON NURSERIES, SHOLTSPAR
059334HOOK E.H. 109 ELMTREE ROAD, GANNET PARK, NORTHAMPTON
073872HOME TAVERN, WESTLEIGH
000931FOREST, IVER, BUCKS
/*
//GO.SYSPRINT DD SYSOUT=A,DCB=(RECFM=V,BLKSIZE=88)
//GO.SORTOUT DD SYSOUT=A,DCB=(RECFM=V,BLKSIZE=88)
//GO.SYSOUT DD SYSOUT=A
//GO.SORTWK01 DD UNIT=SYSDA,SPACE=(CYL,1)
//GO.SORTWK02 DD UNIT=SYSDA,SPACE=(CYL,1)
//*
```

Figure 148 (Part 2 of 2). Example of Sorting Varying Length Records Using Input and Output Handling Routines

CHAPTER 13. CHECKPOINT/RESTART

The PL/I Checkpoint/Restart feature provides a convenient method of taking checkpoints during the execution of a long-running program in a batch environment. It cannot be used in a TSO environment.

At points specified in the program, information about the current status of the program is written as a record on a data set. If the program terminates due to a system failure, this information can be used to restart the program close to the point where the failure occurred, avoiding the need to rerun the program completely.

This restart can be either automatic or deferred. An automatic restart is one that takes place immediately (provided the operator authorizes it when requested by a system message). A deferred restart is one that is performed later as a new job.

You can request an automatic restart from within your program without a system failure having occurred.

PL/I Checkpoint/Restart uses the Advanced Checkpoint/Restart Facility of the operating system. This is fully described in the publication Advanced Checkpoint/Restart.

To use checkpoint/restart you must do the following:

- Request, at suitable points in your program, that a checkpoint record is written. This is done with the built-in subroutine PLICKPT.
- Provide a data set on which the checkpoint record can be written.
- Also, to ensure the desired restart activity, you may need to specify the RD parameter in the EXEC or JOB statement (see the publication JCL Reference).

Note: You should be aware of the restrictions affecting data sets used by your program. These are detailed in the publication Advanced Checkpoint/Restart.

WRITING 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 CALL statement has the form:

```
CALL PLICKPT [(ddname[,check-
     id[,org[,code]]])];
```

The four arguments are all optional. If an argument is not used, it need not be specified unless another argument that follows it in the given order is specified. In this case, the unused argument must be specified as a null string. The following paragraphs describe the arguments.

"ddname" is a character string constant or variable specifying the name of the DD statement defining the data set that is to be used for checkpoint records. If this argument is omitted, the system will use the default ddname SYSCHK.

"check-id" is a character string constant or variable specifying the name that you want to assign to the checkpoint record so that you can identify it later, if required. If this argument is omitted, the system will supply a unique identification and print it at the operator's console.

"org" is a character string constant or variable with the attributes CHARACTER(2) whose value indicates, in operating system terms, the organization of the checkpoint data set. PS indicates sequential (that is, CONSECUTIVE) organization; PO represents partitioned organization. If this argument is omitted, PS is assumed.

"code" is a variable with the attributes FIXED BINARY (31), which can receive a return code from PLICKPT. The return code has the following values:

0 A checkpoint has been successfully taken.

4 A restart has been successfully made.

8 A checkpoint has not been taken. The PLICKPT statement should be checked.

12 A checkpoint has not been taken. Check for a missing DD statement, a hardware error, or insufficient space in the data set. A checkpoint will fail if taken while a DISPLAY statement with the REPLY option is still incomplete or if the program is using multitasking.

16 A checkpoint has been taken, but ENQ macro calls are outstanding and will not be restored on restart. This situation will not normally arise for a PL/I program.

CHECKPOINT DATA SET

A DD statement defining the data set on which the checkpoint records are to be placed, must be included in the job control procedure. This data set can have either CONSECUTIVE or partitioned organization. Any valid ddname can be used. If you use the ddname SYSCHK, you do not need to specify the ddname when invoking PLICKPT.

A data set name need be specified only if you want to keep the data set for a deferred restart. The I/O device can be any magnetic-tape or direct-access device.

If you want to obtain only the last checkpoint record, then specify status as NEW (or OLD if the data set already exists). This will cause each checkpoint record to overwrite the previous one.

If you want to retain more than one checkpoint record, specify status as MOD. This will cause each checkpoint record to be added after the previous one.

If the checkpoint data set is a library, then "check-id" is used as the member-name. Thus a checkpoint will delete any previously-taken checkpoint with the same name.

For direct-access storage, enough primary space should be allocated to store as many checkpoint records as you will retain. You can specify an incremental space allocation, but it will not be used. A checkpoint record is approximately 5000 bytes longer than the area of main storage allocated to the step.

No DCB information is required, but you can include any of the following, where applicable:

```
OPTCD=W, OPTCD=C, RECFM=UT, NCP=2, TRTCH=C
```

These subparameters are described in your JCL manual.

PERFORMING A RESTART

A restart can be automatic or deferred. Automatic restarts can be made after a system failure or from within the program itself. All automatic restarts need to be authorized by the operator when requested by the system.

AUTOMATIC RESTART AFTER A SYSTEM FAILURE

If a system failure occurs after a checkpoint has been taken, the automatic restart will occur at the last checkpoint if you have specified RD=R (or omitted the RD parameter) in the EXEC or JOB statement.

If a system failure occurs before any checkpoint has been taken, then an automatic restart, from the beginning of the job step, can still occur if you have specified RD=R in the EXEC or JOB statement.

After a system failure occurs, you can still force automatic restart from the beginning of the job step by specifying RD=RNC in the EXEC or JOB statement. By specifying RD=RNC, you are requesting an automatic step restart without checkpoint processing should another system failure occur.

AUTOMATIC RESTART FROM WITHIN THE PROGRAM

A restart can be requested at any point in your program. The rules applying to the restart are the same as for a restart after a system failure. To request the restart, you must execute the statement:

```
CALL PLIREST;
```

To effect the restart, the compiler terminates the program abnormally, with a system completion code of 4092. Therefore, to use this facility, the system completion code 4092 must not have been deleted from the table of eligible codes at system generation.

DEFERRED RESTART

To ensure that automatic restart activity is canceled, but that the checkpoints are still available for a deferred restart, specify RD=NR in the EXEC or JOB statement when the program is first executed.

If a deferred restart is subsequently required, the program must be submitted as a new job, with the RESTART parameter in the JOB statement. The RESTART parameter specifies the job step at which the restart is to be made and, if you want to restart at a checkpoint, the name of the checkpoint record. The RESTART parameter has the form:

```
RESTART=(stepname[,check-id])
```

For a restart from a checkpoint, you must also provide, immediately before the EXEC statement for the job step, a DD statement, with the name SYSCHK, defining the data set containing the checkpoint record.

MODIFYING CHECKPOINT/RESTART ACTIVITY

You can cancel automatic restart activity from any checkpoints taken in your program by executing the statement:

```
CALL PLICANC;
```

However, if you have specified RD=R or RD=RNC in the JOB or EXEC statement, automatic restart can still take place from the beginning of the job step.

Also, any checkpoints already taken will still be available for a deferred restart.

You can cancel any automatic restart, and also the taking of checkpoints, even if requested in your program, by specifying RD=NC in the JOB or EXEC statement.

CHAPTER 14. INTERLANGUAGE COMMUNICATION WITH COBOL AND FORTRAN

The PL/I interlanguage facilities permit communication, at execution time, between programs compiled by the PL/I Checkout and Optimizing Compilers and programs compiled by one of the following compilers, and executed using the corresponding library:

OS FORTRAN IV Compiler
(H Extended)

OS FORTRAN Library Mod II

OS/VS COBOL Compiler and
Library
(Library only)

In addition, such programs compiled by the PL/I Optimizing Compiler can communicate with programs compiled by one of the following compilers, and executed using the corresponding library:

VS FORTRAN Compiler and Library
(Library only)

VS COBOL II
Compiler and Library
(Library only)

Communication between a PL/I program and a program compiled by one of the FORTRAN or COBOL compilers can be achieved in two ways:

- By using a common data set for the PL/I and COBOL/FORTRAN routines.
- By invoking a COBOL/FORTRAN routine from a PL/I routine, or vice versa, and by passing data either as arguments or in the form of static storage.

If a common data set is used to communicate between a PL/I and a COBOL routine, the COBOL option of the ENVIRONMENT attribute may be required. Although this option initiates remapping of PL/I structures, it is in no way associated with the interlanguage facilities described here; a file with this option cannot be used as a file argument or a file parameter. For use of the COBOL option of the ENVIRONMENT attribute, see "COBOL Option—Data Interchange" on page 132.

A PL/I procedure can invoke a COBOL routine by use of the CALL statement, or can invoke a FORTRAN routine by use of the CALL statement or a function reference. Alternatively, a PL/I procedure can be invoked by use of the corresponding language features in a COBOL or a FORTRAN main program or routine. Arguments can be passed on invocation, and a value can be returned for function references.

Interlanguage calls to COBOL and/or FORTRAN cannot be made from PL/I FETCHed procedures. COBOL dynamic CALL statements may not be used in an interlanguage environment.

A PL/I procedure cannot invoke a COBOL or a FORTRAN routine as a task. Only one task of a PL/I program can have active COBOL or FORTRAN routines at any one time. If a PL/I program has more than one task active at the same time, then, if one of these tasks has invoked a COBOL or a FORTRAN routine, you must ensure that the other tasks wait until control has returned to the PL/I program before another non-PL/I routine is invoked.

A COMMON block in FORTRAN (other than dynamic common; dynamic common must be passed as arguments) has storage equivalent to that of a STATIC EXTERNAL variable in PL/I. If a COMMON block and a STATIC EXTERNAL variable are given the same name, then they will be allocated the same block of storage, in the same way as two identical STATIC EXTERNAL variables in PL/I. Assigning a value to one variable causes the same value to be assigned to the other. There is no similar equivalence in COBOL—no COBOL variable can have common storage with a PL/I variable other than as an argument or parameter.

The interlanguage facilities are entirely provided by the PL/I compiler; they are obtained by specifying the appropriate language items in the invoking or invoked PL/I procedure. Existing COBOL or FORTRAN programs or routines generally do not need modification or recompiling for interlanguage use; new programs or routines can be written in these languages and compiled as before, without the need to anticipate interlanguage communication. Thus existing COBOL or FORTRAN application programs can be extended by the use of PL/I procedures, while COBOL or FORTRAN libraries can be made available to new or existing PL/I procedures.

In the context of this chapter, "routine" includes a COBOL subprogram, or a FORTRAN subroutine or function, including a FORTRAN library function. The conventions that exist in these languages for handling subroutines and functions apply normally, and are not modified for interlanguage use. In particular, the restriction that a FORTRAN function cannot be invoked without passing an argument or arguments still applies when the invocation is from a PL/I routine.

Facilities are provided to extend PL/I interrupt handling to cover invoked COBOL or FORTRAN routines.

INVOKING COBOL FROM PL/I

If you invoke a COBOL routine after a PL/I on-unit has intercepted an abend, unpredictable results may occur. You should also consider the effects of the INTERRUPT option, as discussed in "INTERRUPT Option" on page 21, when determining what will occur in the event of an abnormal termination.

COBOL programs should specify the COBOL NOSTAE option to allow PL/I to handle abends unless you require COBOL debugging facilities. In that case, specify the PL/I NOSTAE option. If COBOL ESTAE error handling is invoked before PL/I STAE handling, you may encounter unpredictable results.

ARGUMENTS AND PARAMETERS

While a detailed knowledge of COBOL or FORTRAN is not essential for use of the interlanguage facilities, you may need to be aware of the equivalents in data organization in PL/I and the other two languages. These equivalents must be understood in order to achieve argument/parameter matching.

The interlanguage facilities resolve differences in the mapping for equivalent data organizations, when matching arguments and parameters; you can, if you wish, override this action.

PASSING ARGUMENTS TO COBOL OR FORTRAN ROUTINES

When an argument is passed to a COBOL or a FORTRAN routine, the data type is determined in the normal PL/I manner; that is, from the parameter descriptor list of the associated entry declaration, or from the argument itself. The interlanguage facilities ensure, however, that the addressing mechanism for the argument is that used by the invoked language, and unless otherwise required, the mapping of any aggregates passed is that used by the invoked language. Since the interlanguage

facilities provided by PL/I cannot look at the parameter in the invoked routine, it is your responsibility to ensure that the parameter in the invoked routine corresponds in data type and organization to the argument description in PL/I.

If the PL/I compiler can determine, at compile-time, that the mapping of a structure or array argument is the same in PL/I as in the invoked language, the argument is passed directly to the invoked routine. However, where such mapping equivalence does not exist, the interlanguage facilities provide for a dummy argument to be passed, where the dummy is mapped according to the rules of the invoked language. See the section on "Structure Mapping" in the OS and DOS PL/I Language Reference Manual.

If the PL/I data types of arguments passed to FORTRAN or COBOL have no equivalents in these languages, a warning message is produced at compile-time. At execution-time the results are undefined, and may include abnormal termination.

DATA TYPES: PL/I has more data types than either COBOL or FORTRAN; some have no equivalents in these languages. The extent to which PL/I data types have equivalents in COBOL or FORTRAN, and therefore can be passed as arguments, is summarized here.

PROBLEM DATA: Most of the PL/I data types have equivalents in either COBOL or FORTRAN. Tables of data equivalents for PL/I-COBOL and PL/I-FORTRAN are given in Figure 149 on page 346 and Figure 151 on page 353, respectively.
PROGRAM-CONTROL DATA: Arguments of any program-control data type can be passed to an invoked COBOL or FORTRAN routine. An entry argument can be passed and used within the invoked routine, and then only if the routine is a FORTRAN routine. Arguments of any other data type should not be used in the invoked routine except to be passed in turn to a PL/I procedure.

DATA-MAPPING: In order that an argument can be successfully passed to a COBOL or FORTRAN routine, the mapping of the actual argument passed must correspond to the mapping assumed for the parameter by COBOL or FORTRAN.

For an element argument, the only requirement is that the alignments of argument and parameter are compatible. In PL/I the alignment of variables is determined by the ALIGNED and UNALIGNED attributes. The equivalent specifications in COBOL and FORTRAN are:

PL/I	COBOL	FORTRAN
ALIGNED	SYNCHRONIZED	Normal alignment
UNALIGNED	Unsynchronized	No equivalent

The alignment of a PL/I argument is deduced, like the data type, from the parameter descriptor list or from the argument itself. Only ALIGNED elements may be passed to SYNCHRONIZED COBOL parameters, or to FORTRAN parameters. Either ALIGNED or UNALIGNED elements can be passed to COBOL unsynchronized parameters. It is your responsibility to ensure that these alignments are compatible.

The problem is more complicated for data aggregates. A PL/I or a COBOL structure, for example, can have either of the alignment stringencies given above. In addition, each member can have its own alignment stringency or all members can have the same alignment stringency. Padding bytes are inserted by the mapping algorithm for the particular language, in order to preserve the required alignment for each member. In a PL/I structure, the alignments are adjusted, where possible, to minimize the amount of padding required; this adjustment does not occur in a COBOL structure. The result is that a structure mapped with the PL/I mapping algorithm may not have the same layout in main storage as a structure mapped with the COBOL algorithm.

COBOL				PL/I			
		Alignment				Alignment	
Data Type	Length (bytes)	Synch. (aligned)	Unsynch. (unaligned)	Data Type	Length (bytes)	Aligned	Unaligned
COMPUTATIONAL[1] dec length: 1–4	2	Halfword	Byte	FIXED BINARY(15,0) (halfword integer)	2	Halfword	Byte
5–9	4	Fullword	Byte	FIXED BINARY(31,0) (fullword integer)	4	Fullword	Byte
10–18	8	Fullword	Byte	No equivalent	–	–	–
COMPUTATIONAL-1[3]	4	Fullword	Byte	FLOAT DEC(6) (short float)	4	Fullword	Byte
COMPUTATIONAL-2[3]	8	Doubleword	Byte	FLOAT DEC(16) (long float)	8	Doubleword	Byte
COMPUTATIONAL-3	1[2]	Byte	Byte	FIXED DEC	1[2]	Byte	Byte
DISPLAY	any	Byte	Byte	CHARACTER	any	Byte	Byte

[1] Decimal length is equal to the number of 9s in the picture.
[2] The length of 1 byte applies to the smallest fixed decimal value (i.e., 1 digit). For other values, the length is given by CEIL((number of digits + 1)/2) bytes.
[3] OS VS COBOL Release 2.4 only, not VS COBOL II.

Figure 149. COBOL—PL/I Data Equivalents

Similarly, the mapping of arrays is different in PL/I and FORTRAN. PL/I stores arrays of more than one dimension in row-major-order, while FORTRAN stores them in column-major-order. Hence, for arrays with more than one dimension, a reference to an element in PL/I is obtained by reversing the order of the subscripts that would be used in FORTRAN to refer to the same element.

The interlanguage facilities resolve these problems by creating dummy arguments for PL/I data aggregates passed as arguments to COBOL or FORTRAN routines. When a PL/I ALIGNED structure is passed as an argument to a COBOL routine, the algorithm used for mapping the argument in both languages is considered. If the compiler can determine that the mappings are identical, the argument is passed directly to the COBOL routine.

However, if the compiler cannot determine that the mappings are identical, a dummy argument is created, mapped according to the COBOL SYNCHRONIZED mapping algorithm. The values of the members of the PL/I structure are assigned to the corresponding members in the dummy argument; the dummy is then passed as an argument to the COBOL routine. On return to the PL/I procedure, the values in the dummy argument (which may or may not have been

changed) are assigned to the corresponding members of the original PL/I argument.

Similarly, when a PL/I array is passed as an argument to a FORTRAN routine, the mapping of the array in both languages is considered. If the arrays are unidimensional, and are in connected storage and are aligned identically, the argument is passed directly to the invoked FORTRAN routine. If either the arrays are unidimensional and do not meet the above conditions, or are multidimensional, a dummy argument is created and mapped according to FORTRAN array handling. (In effect, this means the subscripts are reversed.) The values of the PL/I array elements are assigned to the corresponding elements in the dummy argument. The dummy is then passed as an argument to the FORTRAN routine. On return to the PL/I procedure, the values in the dummy argument (which may or may not have been changed) are assigned to the appropriate elements of the PL/I argument.

You can specify certain options that inhibit or restrict the effect of the interlanguage facilities for remapping data aggregates. If several are passed at an invocation, you can, for example, inhibit the facilities for one argument, allow them for another argument, or restrict them for a third argument.

INVOKING COBOL OR FORTRAN ROUTINES

Invocation of a COBOL or FORTRAN routine is performed by a CALL statement or (in the case of a FORTRAN routine only) function reference that specifies an entry constant or variable whose value corresponds to the entry point of a COBOL or FORTRAN routine. The entry point must not be that of a FORTRAN main program. The entry constant or variable must be identified as invoking COBOL or FORTRAN by use of the appropriate options in the OPTIONS attribute in the declaration of the entry in the PL/I program. You may also specify, in this declaration, options that suppress remapping of data aggregates and an option that allows PL/I to deal with certain interrupts in the COBOL or FORTRAN routine.

The options are:

COBOL
This specifies that the designated entry point is in a COBOL routine.

FORTRAN
This specifies that the designated entry point is in a FORTRAN routine.

NOMAP
This specifies that a dummy argument is not created; the aggregate argument is passed directly to the invoked routine.

NOMAPIN
This specifies that, if a dummy argument is created, it is not initialized with the values of the aggregate argument.

NOMAPOUT
This specifies that, if a dummy argument is created, then, on return, the values in the dummy argument are not assigned to the aggregate argument.

INTER
This specifies that any interrupts occurring during the execution of a COBOL or FORTRAN routine that are not dealt with by the COBOL or FORTRAN interrupt-handling facilities are dealt with by the PL/I interrupt-handling facilities (see also "Handling Interrupts" on page 357).

The NOMAPIN and NOMAPOUT options should be used if initialization is not required whenever program efficiency is important, because they allow the compiler to omit unnecessary initialization code.

ARGn
This is an option of NOMAP, NOMAPIN, and NOMAPOUT that specifies which arguments the option applies to. If no ARGn is specified, the option is applied to all arguments.

The following points should be noted in the declaration of the entry name:

- Either COBOL or FORTRAN (but not both) can appear in the declaration. One or more of the options NOMAP, NOMAPIN and NOMAPOUT can appear in the same declaration.
- The RETURNS attribute cannot be used with the COBOL option, as COBOL does not provide function subprograms.
- An entry variable or a parameter can be declared with the interlanguage options.
- An entry name with the interlanguage options can appear in a GENERIC attribute specification.
- The entry constant name of the COBOL or FORTRAN routine may have 1 through 8 characters. If more than 8 characters are specified, only the leftmost 8 are taken.
- Specifying NOMAPIN and NOMAPOUT for the same argument is equivalent to specifying NOMAP for that argument; that is, no dummy argument is created.
- NOMAP, NOMAPIN, and NOMAPOUT are effective only for structures passed to COBOL and arrays passed to FORTRAN.

Examples

1.
```
DCL COBOL ENTRY (CHAR(5))
    OPTIONS(COBOL INTER),

  COBOLB ENTRY (1, 2 FIXED, 2 FLOAT)
    OPTIONS(COBOL NOMAPIN),

  COBOLBX  OPTIONS(COBOL) EXTERNAL
    ENTRY(...);
```

2.
```
DCL FORTA ENTRY(FIXED BINARY)
      OPTIONS(FORTRAN) RETURNS
      (FLOAT (5));
```

3.
```
DCL A EXTERNAL ENTRY(...) VARIABLE
      OPTIONS (FORTRAN),

    B OPTIONS(FORTRAN);
          .
          .
          .
     A=B;
     CALL A(...);
```

4.
```
DCL A GENERIC (COBOLZ
      WHEN(CHARACTER),
        FORTZ WHEN(FIXED BINARY)),

    COBOLZ OPTIONS(COBOL),

    FORTZ OPTIONS(FORTRAN);
```

```
5.  DCL A ENTRY;

    CALL X(A);
           .
           .
           .
    X: PROC(B);
    DCL B OPTIONS(COBOL);

6.  DCL COBSUB ENTRY(...,...,...,)
        OPTIONS(COBOL,NOMAP(ARG1,ARG3));
           .
           .
           .
        CALL COBSUB(A,B,C);
           .
           .
           .
        CALL COBSUB(X,Y,Z);
```

PASSING ARGUMENTS FROM COBOL OR FORTRAN ROUTINES

When an argument is passed to a PL/I procedure from COBOL or FORTRAN, the data type is determined in the normal PL/I manner; that is, from the declaration of the parameter. The interlanguage facilities ensure that the addressing mechanism used for the parameter is that used by PL/I, and that, unless otherwise required, the mapping of any aggregate parameters passed is also that used by PL/I. Since the interlanguage facilities provided by PL/I cannot look at the argument in the routine invoking PL/I, it is your responsibility to ensure that the argument passed to PL/I corresponds in data type and organization to the parameter declared in PL/I.

Data Mapping

The situation is similar to that which occurs on invocation of COBOL or FORTRAN by PL/I. The mapping of the argument on entry to the PL/I procedure must correspond to the mapping used by PL/I in addressing the parameter.

For element arguments and parameters, this means that a FORTRAN or a synchronized or unsynchronized COBOL argument may be passed to an UNALIGNED PL/I parameter, or that a synchronized COBOL argument or a FORTRAN argument can be passed to an ALIGNED PL/I parameter.

For aggregate arguments and parameters where the mapping of the argument in COBOL (synchronized) or FORTRAN differs from the mapping of the parameter in PL/I, the interlanguage facilities resolve the problem by creating a dummy argument which is passed to the PL/I procedure.

The dummy argument is mapped according to PL/I rules, and, before invocation of the PL/I procedure, the values of the members of the COBOL or FORTRAN argument are assigned to the corresponding members of the dummy argument. On return from the PL/I procedure, the values of the members of the dummy argument are assigned back to the original argument.

If the compiler can recognize that the mapping in COBOL or FORTRAN and PL/I are equivalent, no such dummy is created.

Alternatively, you can inhibit the creation of the dummy, or the assignments between the original argument and the created dummy, by means of options.

INVOKING PL/I ROUTINES FROM COBOL OR FORTRAN

The entry points in a PL/I procedure that are to be invoked from COBOL or FORTRAN must be identified by the appropriate options in the corresponding PROCEDURE or ENTRY statement. You may also specify options that suppress remapping of data aggregates.

Because of the way the PL/I environment is preserved, the COBOL DYNAM and ENDJOB options may not be used when invoking a PL/I PROCEDURE.

COBOL
This specifies that the entry point can only be invoked by a COBOL routine.

FORTRAN
This specifies that the entry point can only be invoked by a FORTRAN routine.

NOMAP
This specifies that a dummy argument is not created; the COBOL or FORTRAN aggregate argument is passed directly to PL/I.

NOMAPIN
This specifies that, if a dummy argument is created, it is not initialized with the values of the aggregate argument.

NOMAPOUT
This specifies that, if a dummy argument is created its values are not assigned back to the aggregate argument on return. The NOMAPIN and NOMAPOUT options should be used, if initializations are not required, whenever program efficiency is important, since they allow the compiler to omit unnecessary initialization code.

Parameter list
The parameter or parameters to which the NOMAP, NOMAPIN, or NOMAPOUT options apply can be specified in a list. If no list is specified, the option is applied to all parameters.

The following points should be noted when coding the PROCEDURE or ENTRY statement:

- Only one of the options MAIN, COBOL, or FORTRAN can appear in the same statement. One or more of the options NOMAP, NOMAPIN, or NOMAPOUT can appear in the same statement.
- If the parameters for the procedure include strings, areas, or arrays; the lengths, sizes, or bounds for these must be specified as integers.
- The RETURNS option cannot be specified for any entry point invoked by a COBOL routine.
- Specifying NOMAPIN and NOMAPOUT for the same argument is equivalent to specifying NOMAP for that argument; that is, no dummy argument is created.
- NOMAP, NOMAPIN, and NOMAPOUT are effective only for structures passed from COBOL and arrays passed from FORTRAN.

Examples:

```
1. P1: PROC(A,B,C) OPTIONS(FORTRAN
   NOMAPIN(C) NOMAPOUT(A));
        DCL A(3,4) FLOAT BIN(20),
            B FIXED BIN(31),
            C(5,6) FLOAT DEC(6);

2. P2: PROC(R,S,T) OPTIONS (FORTRAN
        NOMAP);
```

```
3. P3: PROC(X,Y) OPTIONS(COBOL NOMAPIN(X)
   NOMAPOUT(Y));
        DCL 1 X,  2...2...3...,
            1 Y,  2...2...3...;
```

MATCHING COBOL ARGUMENTS/PARAMETERS

Argument/parameter matching across a PL/I-COBOL interface requires a knowledge of the equivalence of data types and of data organization in the two languages. The PL/I equivalents of the COBOL data types are shown in Figure 149 on page 346. These are the PL/I data types that should appear in PL/I parameter descriptors associated with COBOL arguments or parameters, respectively.

While a knowledge of equivalent data types is sufficient for specifying COBOL items in terms of PL/I element variables, the specification of equivalent data aggregates (group items in COBOL, structures or arrays in PL/I) requires a knowledge of the data-organization descriptions of the two languages. The example given in Figure 150 on page 352 shows how a COBOL data aggregate is described in PL/I terms.

In COBOL, the OCCURS clause cannot be nested to more than three levels. This imposes a restriction on any PL/I array within a structure passed as an argument to a COBOL routine. Also, the OCCURS clause cannot appear on a level-01 entry. This precludes the use of a level-01 array in a PL/I structure passed to or from a COBOL routine.

A PL/I structure that contains an area or a bit variable should not be passed as an argument to a COBOL routine. If it is, a diagnostic message is produced and the structure is not remapped.

A bit or character string with the VARYING attribute may be passed to a COBOL routine, although there is no equivalent attribute in COBOL. The address of the start of the 2-byte length prefix is passed, so that the prefix constitutes the first 2 bytes of the COBOL string. Conversely, when COBOL data is passed to a PL/I string parameter with the VARYING attribute, the first 2 bytes of the argument form the parameter's length prefix.

MATCHING FORTRAN ARGUMENTS/PARAMETERS

Argument/parameter matching across a PL/I-FORTRAN interface, and the use of common storage for PL/I and FORTRAN variables, requires a knowledge of the equivalence of data types and of data organizations in the two languages. The PL/I equivalents of the FORTRAN data types are shown in Figure 151 on page 353. These are the PL/I data types that should appear in PL/I parameters or parameter descriptors associated with FORTRAN arguments or parameters respectively, and in the declaration of STATIC EXTERNAL variables with the same names as FORTRAN COMMON blocks.

Specification of equivalent data aggregates in PL/I and FORTRAN is simpler than in PL/I and COBOL, as the only data aggregates that exist in FORTRAN are arrays. Problems arise when using unconnected unidimensional arrays or multidimensional arrays as PL/I arguments.

Generally, when passing arguments between PL/I and FORTRAN, the interlanguage facilities pass a unidimensional array directly to the invoked routine, without the creation of a dummy argument. However, if a PL/I unidimensional array in unconnected storage is passed as an argument to a FORTRAN routine, the interlanguage facilities create a dummy argument into which the unconnected array is mapped. The dummy is then passed as the argument. On return, the values in the dummy are assigned to the corresponding elements in the array.

A dummy argument is always created for a multidimensional array passed between PL/I and FORTRAN routines, unless the NOMAP option is specified.

COBOL

```
01  A.
    02  B OCCURS 3 TIMES.
        03  C OCCURS 4 TIMES.
            04  D OCCURS 5 TIMES USAGE COMP-3
                PIC S9999V999.
    02  E USAGE DISPLAY.
        03  F PIC X(8).
        03  G PIC 9(8).
    02  DUMMY OCCURS 6 TIMES.
        03  H OCCURS 7 TIMES USAGE COMP
                PIC S9999 SYNCHRONIZED.
```

PL/I

```
1 A ALIGNED,
  2 B(3),
    3 C(4),
      4  D(5) FIXED DECIMAL(7,3),
  2 E,
    3 F CHAR(8),
    3 G PIC '(8)9',
  2 H(6,7) FIXED BINARY(15,0);
```

Figure 150. Declaration of a Data Aggregate in COBOL and PL/I

FORTRAN			PL/I			
					Alignment	
Data Type	Length (bytes)	Alignment[1]	Data Type	Length (bytes)	Aligned	Unaligned
INTEGER*2	2	Halfword	REAL FIXED BINARY(15,0)	2	Half-word	Byte
INTEGER*4	4	Fullword	REAL FIXED BINARY(31,0)	4	Full-word	Byte
REAL*4	4	Fullword	REAL FLOAT DEC(6) (real short float)	4	Full-word	Byte
REAL*8	8	Doubleword	REAL FLOAT DEC(16) (real long float)	8	Double-word	Byte
REAL*16	16	Doubleword	REAL FLOAT DEC(33) (real extended float)	16	Double-word	Byte
COMPLEX*8	8	Fullword	COMPLEX FLOAT DEC(6) (complex short float)	8	Full-word	Byte
COMPLEX *16	16	Doubleword	COMPLEX FLOAT DEC(16) (complex long float)	16	Double-word	Byte
COMPLEX *32	32	Doubleword	COMPLEX FLOAT DEC(33) (complex extended) float)	32	Double-word	Byte
LOGICAL*1	1	Byte	BIT(8)	1	Byte	Bit[2]
LOGICAL*4	4	Fullword	BIT(32)	4	Byte	Byte

[1]Generally, FORTRAN data is held in main storage with these alignments. COMMON data, however, is always byte-aligned. This could cause a specification interrupt if the items in the COMMON area are not stored in order of decreasing stringency.

[2]The fact that the alignment required of unaligned bit strings is bit rather than byte does not affect PL/I-FORTRAN data interchange, since the FORTRAN string will always take up an integral number of bytes.

Figure 151. FORTRAN-PL/I Data Equivalents

If a PL/I array of bit strings is passed as an argument to a FORTRAN routine, only 8 or 32 should be specified for the string lengths. If values other than these are specified, a diagnostic message is produced and the array is not remapped. Similarly, only these lengths should be used for PL/I variables having storage common with FORTRAN variables.

COMPILE-TIME RETURN CODES

As part of the interlanguage facilities of PL/I, diagnostic messages are produced, and the return code is set appropriately, if you specify arguments or parameters whose attributes are such that errors may occur at execution time. The compiler never prevents data being passed, nor does it attempt to correct errors; although it produces messages to indicate likely sources of error, it always allows you to attempt to pass any type of data you specify.

Figure 152 on page 354 shows the return codes generated by various types of PL/I data.

For further information on compile-time return codes, see Figure 11 on page 55.

PL/I Attribute	COBOL Argument	COBOL Parameter	FORTRAN Argument	FORTRAN Parameter
ALIGNED	0000	0000	0000	0000
AREA	Note 1	Note 1	Note 1	Note 1
BINARY	0000	0000	0000	0000
BIT	Note 1	Note 1	Note 2	Note 2.
CHARACTER	0000	0000	0004	0004
COMPLEX	0004	0004	Note 4	Note 4
CONNECTED	0000	0000	0000	0000
CONTROLLED	0000	0012	0000	0012
DECIMAL	0000	0000	Note 3	Note 3
DEFINED	0000	-	0000	-
Dimension	Note 8	Note 8	0000	0000
ENTRY	0004	0004	0004	0004
EVENT	0004	0004	0004	0004
FILE	0004	0004	0004	0004
FIXED	0000	0000	0000	0000
FLOAT	0000	0000	0000	0000
LABEL	0004	0004	0004	0004
OFFSET	0004	0004	0004	0004
PICTURE	0000	0000	0004	0004
POINTER	0004	0004	0004	0004
Precision	Note 6	Note 6	Note 7	Note 7
REAL	0000	0000	0000	0000
Structure	0000	0000	Note 1	Note 1
TASK	0004	0004	0004	0004
UNALIGNED	Note 9	0000	Note 9	0000
Unconnected	Note 5	0000	Note 5	0000
VARYING	0004	0004	0004	0004

Figure 152. Return Codes Produced by PL/I Data Types

Notes to Figure 152:

1 Checkout compiler: 0004
Optimizing compiler: 0008
In both cases, creation of a dummy argument is suppressed.

2 BIT(8) or BIT(32): 0000
Any other length: 0008
In latter case, creation of a dummy argument is suppressed.

3 FLOAT DECIMAL: 0000
FIXED DECIMAL: 0004

4 FLOAT COMPLEX: 0000
FIXED COMPLEX: 0008

5 If creation of temporary suppressed by NOMAP option: 0012
If no NOMAP option: 0000

6 Variable is FIXED (p,0), or is short or long FLOAT: 0000
Variable is BINARY FIXED (p,q) with q¬=0 or is extended FLOAT: 0004

7 Variable is FLOAT, or is FIXED BINARY with precision (p,0): 0000
Variable is FIXED DECIMAL, or is BINARY (p,q) with q¬=0: 0004

8 If item is element of a structure or is a minor structure: 0000
All other cases: 008

9 If argument is an aggregate and creation of temporary is suppressed by NOMAP, or if argument is scalar: 0012
If argument is an aggregate and no NOMAP: 0000

USING COMMON STORAGE

A variable in a PL/I program can be allocated the same block of storage as a group of variables in a FORTRAN routine. This storage can then be used to communicate between the two routines. Allocation of common storage is achieved by declaring a PL/I variable to be STATIC EXTERNAL and to have the same name as a COMMON block in the FORTRAN routine. The STATIC EXTERNAL variable and the COMMON block will then be equivalent to two declarations of a STATIC EXTERNAL variable in different external PL/I procedures. The number of variables using common storage is not limited to two. Any number of identical STATIC EXTERNAL variables in different PL/I procedures may be used together with any number of identical COMMON blocks in different FORTRAN routines, if all the procedures and routines are link-edited into a single program. Methods of link-editing are given in Chapter 3, "The Linkage Editor and the Loader" on page 65.

The STATIC EXTERNAL variables must follow the normal PL/I rules relating to these attributes, and they must be of a data type that corresponds to the data type of the COMMON variables (see Figure 151 on page 353 for a table of corresponding data types). Also, the PL/I variables must be aligned to meet the requirements of the corresponding FORTRAN data type.

The PL/I variables may be initialized using the INITIAL attribute, and the FORTRAN variables may be initialized using a block data subprogram. If the PL/I variables on the one hand and the FORTRAN variables on the other are not initialized to the same value, the procedure or routine encountered first by the linkage-editor determines the initial value of all the variables. It is not an error to initialize a PL/I variable to a different value from a corresponding FORTRAN variable, or to initialize one and not the other.

The PL/I variable may have further variables overlaid upon it by means of the DEFINED attribute, provided that the defined variable meets the data type and alignment requirements of the FORTRAN variable. If the requirements are not met, execution errors may occur.

Common storage cannot be used for a PL/I variable and a COBOL variable. The only facility provided by PL/I for communication of data between a PL/I procedure and a COBOL routine is that for passing arguments.

INTERLANGUAGE ENVIRONMENT

For a program to be executed, a suitable environment must first be established. If the program contains a PL/I main procedure, the PL/I environment is established when the program is first entered. If the main routine is COBOL or FORTRAN, the interlanguage facilities will establish the required PL/I environment when necessary. This section describes the conventions and restrictions in the interlanguage context.

ESTABLISHING THE PL/I ENVIRONMENT

If the main routine of the program is a PL/I main procedure, the PL/I environment is established on entry to the program. Even if this program contains a mixture of PL/I and COBOL or FORTRAN routines, the normal rules for freeing PL/I storage and closing PL/I files apply.

If the main routine of the program is not a PL/I main procedure, the PL/I environment is established when the first PL/I procedure is invoked. The extent of this environment includes the routine that invoked the PL/I procedure (see Figure 153), and the environment remains in existence until that routine is terminated. The environment can be re-established and terminated as frequently as required. Whenever the PL/I environment is destroyed, all PL/I controlled and based storage is released, and all PL/I files are closed.

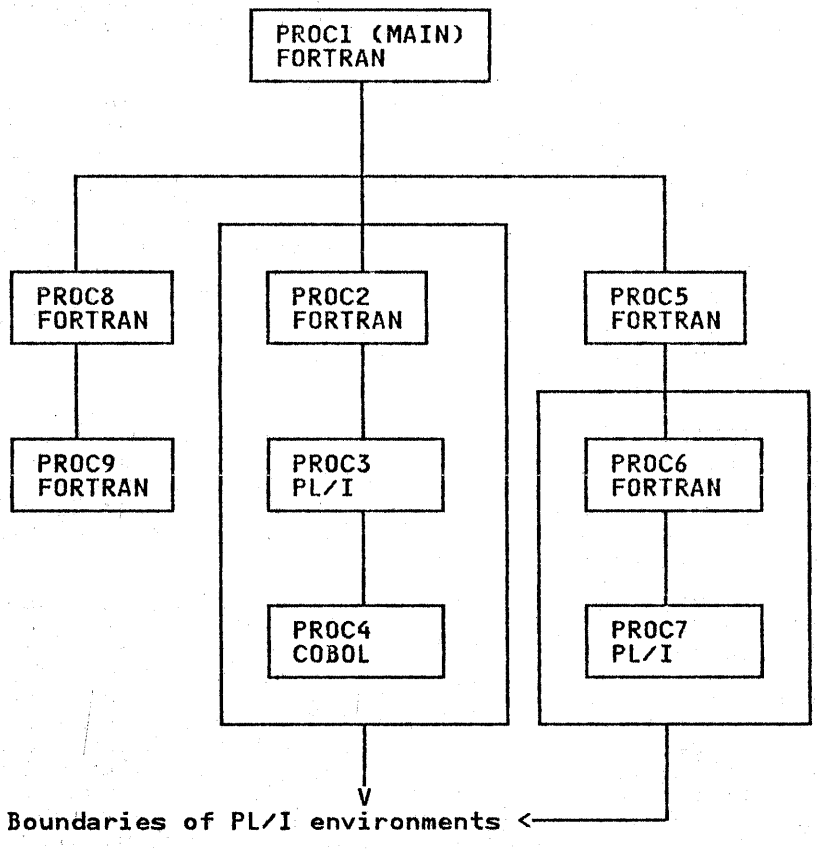


Figure 153. Extent of PL/I Environment

For reasons of efficiency and of programming convenience, the PL/I environment should be destroyed as infrequently as possible during execution of a program. This can be ensured if the main routine is a PL/I main procedure, or if a PL/I procedure, no matter what it contains, is invoked from the main routine. The latter alternative, however, has the disadvantage that if the main routine is in FORTRAN, the PL/I environment will not be ended normally when the final FORTRAN RETURN is executed to return control to the operating system (see "Terminating FORTRAN and COBOL Routines" on page 359).

Note, however, that there must not be two concurrent PL/I environments; this means, for example, that a COBOL program may not call two PL/I main procedures.

ESTABLISHING THE FORTRAN ENVIRONMENT

Before a FORTRAN routine can be executed, a suitable environment must be established. The extent of this environment includes the PL/I procedure that invokes the FORTRAN routine, and this environment remains in existence until the PL/I procedure is terminated.

On each call to FORTRAN a test is made to determine whether a FORTRAN environment has been established. If it has not, FORTRAN initialization routines are invoked. Among other housekeeping tasks performed, the message file, FT06F001, is opened in preparation for FORTRAN error handling. When the FORTRAN environment is terminated, the file is closed.

Since the FORTRAN environment remains in effect only as long as the invoking PL/I procedure is active, considerable overhead can accrue opening and closing FT06F001, if the invoking PL/I program is itself invoked repeatedly.

For reasons of efficiency then, the FORTRAN environment should be destroyed as infrequently as possible during the execution of a program. If VS FORTRAN subroutines or functions are used, VS FORTRAN cannot be called for initialization a second time after it has gone through termination. This is ensured if the PL/I procedure that calls the FORTRAN routine is not terminated until all the FORTRAN calls have been executed, or if the FORTRAN environment is extended to include the outer PL/I procedure by invoking a FORTRAN routine (no matter what it contains, a "RETURN" statement is sufficient) from the outer PL/I procedure.

HANDLING INTERRUPTS

COBOL and FORTRAN routines handle certain hardware interrupts that may occur during their execution, but there are some that they do not handle. The interlanguage communication facilities of PL/I allow any interrupt not dealt with by a COBOL or FORTRAN routine to be handled by any PL/I procedure from which that routine is dynamically descendent.

Specify the INTER option of the OPTIONS attribute when declaring the COBOL or FORTRAN entry name. (See also the INTER option under "Invoking COBOL or FORTRAN Routines" on page 347.) This allows the interrupts not dealt with by the invoked COBOL or FORTRAN routine to be handled by either a PL/I on-unit or by PL/I implicit action. (Except that PL/I cannot handle a ZERODIVIDE interrupt in a division of COMPUTATIONAL-3 data in a routine compiled by a COBOL compiler. Such an interrupt will cause termination of the program.) In PL/I, an on-unit, while established, applies not only to the procedure in which it was created, but also to all procedures that are dynamically descendent from it. If there occurs, during the execution of a COBOL or FORTRAN routine, an interrupt that will not be handled by that routine, and if the routine was invoked by a PL/I procedure in which the INTER option was specified for the COBOL or FORTRAN entry name, then a search is made through all invoking procedures for an appropriate on-unit. If none is

found, implicit action for the condition is taken. If INTER is not specified, no search is made, and the interrupt is dealt with by the operating system control program.

The search passes through all routines in the invoking chain, as far as the limit of the PL/I environment. It is, therefore, possible for the search to include COBOL and FORTRAN routines. Such routines have no effect on the results of the search, since only PL/I on-units are searched for. The operating system may convert some interrupts to an abend, and a PL/I ERROR on-unit may get control to process the abend.

GO TO STATEMENT

The GO TO statement must not be used to transfer control across more than one interlanguage boundary, where an interlanguage boundary is defined as an invocation in which one routine calls another of a different language. Such transfers of control may be initiated inadvertently if you use a GO TO statement in an on-unit. (Execution of a statement that causes entry to an on-unit is not considered as transferring control outside the block or routine. The on-unit may be regarded as being appended to the procedure or routine from which it is entered. This applies even if the on-unit is entered from a COBOL or FORTRAN routine.) Consider the following example:

```
P: PROCEDURE;
 DECLARE LAB LABEL(L1,L2) EXTERNAL,
 FORT ENTRY OPTIONS(FORTRAN INTER);
   ON ERROR GO TO LAB;
     .
     .
 CALL FORT;
     .
     .
 L1: .......;
     .
     .
 END P;

 Q: PROCEDURE OPTIONS(FORTRAN);
    DECLARE LAB LABEL(L1,L2)
    EXTERNAL;
        .
        .
    L2: .......;
        .
        .
 END Q;
```

Assume that the CALL FORT; statement is executed, and that FORT then calls Q. Assume further that an error occurs in Q which initiates entry to the on-unit established in P. At this stage control is still with procedure Q, because the on-unit is regarded as being appended to the procedure from which it was entered. If LAB has the value L1, then the GO TO branch is in error, because it transfers control back to procedure P and in doing so crosses the interlanguage boundaries between Q and FORT and between FORT and P. If LAB has the value L2, the GO TO is not in error because control remains in procedure Q. If an interrupt in FORT caused the on-unit to be entered before Q was called, then the GO TO would not have been in error, if LAB had the value L1; only one interlanguage boundary would be crossed, namely the FORTRAN-PL/I boundary between FORT and P. (LAB should not have the value L2 in this case, because procedure Q is not active.)

TERMINATING FORTRAN AND COBOL ROUTINES

A routine may be terminated by either executing a statement that terminates the whole program, or by handing control back to the calling routine.

The statements that terminate the whole program are STOP in FORTRAN and STOP RUN in COBOL. They are equivalent to the PL/I STOP statement. The effects of these statements are unchanged in a mixed language program; they still terminate the whole program.

If a FORTRAN STOP is executed in a routine within a PL/I environment, that environment is not ended in the normal way. If a COBOL STOP RUN is executed in a routine within a PL/I environment, that environment is ended in the normal way only if it includes the main routine of the program; otherwise the termination is abnormal. The main difference, from your point of view, between a normal and an abnormal ending is that open files in PL/I procedures are not closed in an abnormal ending. This could cause output data to be lost. Considering the example in Figure 153 on page 356, a STOP in PROC2 or a STOP RUN in PROC4 would not close any files that may be open in PROC3, and a STOP in PROC6 would not close any files in PROC7.

A RETURN executed in a FORTRAN subroutine or function that is inside a PL/I environment and which returns control to a routine outside that environment, ends the PL/I environment and causes all files in dynamically descendent PL/I procedures to be closed (in other words, a RETURN statement in a FORTRAN routine that directly invokes a PL/I routine, but which is not dynamically descendent from any PL/I routine). However, a RETURN statement in a FORTRAN main routine is effectively a STOP statement; control is passed to the operating system without any files being closed.

When a COBOL main routine within a PL/I environment returns control to the operating system, the environment ends normally.

EXECUTION-TIME RETURN CODES

The value of the PL/I return code may be set in a PL/I routine by means of the PLIRETC built-in subroutine (see "Execution-time Return Codes" on page 290).

The return code of a non-PL/I routine may be obtained by declaring the entry point with OPTIONS(RETCODE). This option causes the value of the PL/I return code to be set to the value returned by the non-PL/I routine in the lower half of register 15.

The latest value of the PL/I return code can be read by means of the PLIRETV built-in function.

For example:

```
DECLARE AR ENTRY OPTIONS(COBOL,RETCODE);
    .
    .
CALL AR;
IF PLIRETV() = 0 THEN.......;
```

CHAPTER 15. USING PL/I ON CICS

PL/I can be used in conjunction with CICS facilities to write application programs (transactions) for CICS/OS/VS. When this is done, CICS provides facilities to the PL/I program that would normally be provided directly by the operating system. These facilities include most data management facilities and all job and task management facilities.

With PL/I Release 4.0, PL/I macro-level and command-level application programs (transactions) running under CICS use 24-bit addressing mode. These transactions cannot use the 31-bit addressing capabilities of the MVS/XA Operating System.

With OS PL/I Version 1 Release 5.1, PL/I command level application programs (transactions) running under CICS are able to use the 31-bit addressing capabilities of the MVS/XA Operating System. Release 5.1 programs will remain compatible for non-MVS/XA systems.

PL/I transactions can reside above 16 megabytes and address data above or below 16 megabytes. They can ALLOCATE BASED and CONTROLLED variables above 16 megabytes by using the HEAP execution-time option.

Release 5.1 provides support for PL/I programs using the CICS Command-Level Interface (in both 24-bit and 31-bit addressing mode) and CICS Macro-Level Interface (in 24-bit addressing mode only) to run with CICS/OS/VS Version 1 Release 6.1 with upgrade and subsequent releases.

This chapter describes the PL/I-supplied PL/I-CICS interface, and the restrictions and features that apply to PL/I programs compiled on Release 5.1 of the OS PL/I Optimizing Compiler, link-edited with Release 5.1 of the PL/I Resident Library, and executed with Release 5.1 of the PL/I transient libraries, under CICS/OS/VS Release 1.6.1.

NOTE: CICS Release 1.6.1 with upgrade (PTF UP90207 and PTF UP90208) is required in order to run PL/I application programs using Release 5.1 of the PL/I Optimizing Compiler or Libraries. This upgrade supports the functional characteristics of OS PL/I Release 5.1.

PL/I restrictions in the CICS/OS/VS environment when the PL/I-supplied PL/I-CICS/OS/VS interface is in use are shown in Figure 154 on page 361.

PL/I RESTRICTIONS UNDER CICS

Input/Output

OPEN/CLOSE	Only for SYSPRINT
Record I/O	No record I/O statements are allowed.
Stream Input	No stream input is allowed.
Stream Output	No stream output is allowed except to the SYSPRINT file. This is intended for debugging purposes only and, for performance reasons, should not be included in production programs.
DISPLAY	The DISPLAY statement cannot be used.
DELAY	The DELAY statement cannot be used.
DATE	The DATE built-in function cannot be used.
TIME	The TIME built-in function cannot be used.

Other Statements

STOP	The STOP statement cannot be used.
WAIT	The WAIT statement cannot be used.
FETCH	The FETCH statement cannot be used.
RELEASE	The RELEASE statement cannot be used.

Multitasking

No PL/I tasking statements are allowed.

COMPLETION, STATUS, and PRIORITY built-in functions can not be used.

PRIORITY, TASK, and EVENT options can not be used.

Interlanguage Communication

No communication with FORTRAN or COBOL using PL/I's interlanguage facilities is allowed.

Execution-Time Options

Execution-time options can only be specified in the PLIXOPT string.

Specifying the SPIE option has no effect.

ISASIZE, ISAINC and HEAP sizes are limited under CICS.
See "Execution-Time Options" on page 370.

Built-In Subroutines

PLISRT, PLICKPT, and PLICANC cannot be used.

PLIDUMP has certain restrictions and additional functions. See "Use of PLIDUMP" on page 375.

PLIRETC and PLIRETV can be used to communicate between user-written programs link-edited together, but not to communicate with CICS.

Figure 154 (Part 1 of 2). Restrictions on PL/I when Used with CICS

Debugging Facilities

FLOW, COUNT, REPORT, GONUMBER, GOSTMT, and CHECK/NOCHECK can all be used without restriction under the CICS command-level interface, but are subject to restrictions under the macro-level interface.
See pages 371 and 370.

External Calls

External calls to other PL/I routines or to assembler language routines declared with OPTIONS (ASSEMBLER) can be made without restriction. Called subroutines can invoke CICS services, provided the appropriate CICS control blocks were passed to them by their callers.

Floating Point Arithmetic

Floating point arithmetic is usable without restriction, except that extended precision floating point is not supported.
- Floating point registers are saved and restored by the PL/I library where necessary.
- Floating point registers are printed by PLIDUMP.
- Floating point overflow and underflow can be handled in OVERFLOW and UNDERFLOW on-units. The program mask is set for PL/I and CICS/OS/VS, respectively, as appropriate.

Variable Names

Names for variables used in CICS/OS/VS macros cannot exceed 8 characters.

Object Program Size

The load module resulting from a PL/I application program must not occupy more than 524,152 bytes of main storage, except that an RMODE=ANY program on MVS/XA can be up to 16 megabytes in length (although this is not recommended).

Static Storage

Static storage is not alterable if reentrancy is to be maintained.

Static External Variables

STATIC EXTERNAL variables must have the INITIAL attribute because CICS/OS/VS cannot handle common CSECTs.

Figure 154 (Part 2 of 2). Restrictions on PL/I when Used with CICS

In addition to the restrictions listed in Figure 154 on page 361, the following topics are discussed in this chapter:

- PL/I-supplied vs. CICS-supplied interface
- PL/I-CICS transactions
- Macro-level interface
- Command-level interface
- Compatibility
- PL/I storage
  - Lifetime of storage acquired from CICS/OS/VS
  - Storage classes
  - CONTROLLED storage
- "Read-Only" PL/I-CICS transactions
- Output to SYSPRINT
- CHECK and PUT DATA
- Execution-time options
- Error handling
- Use of PLIDUMP
- Interlanguage communications - OPTIONS (ASSEMBLER)
- STORAGE and CURRENTSTORAGE
- PL/I program termination
- PL/I shared library
- Link-editing PL/I-CICS applications.

PL/I-SUPPLIED VS. CICS-SUPPLIED INTERFACE

In the early versions of OS CICS-Standard and CICS/OS/VS (prior to CICS/OS/VS Release 1.5), CICS itself provided an interface between your PL/I program and CICS. This interface consisted of modified PL/I library modules that requested such services as the acquisition and release of storage from CICS. This interface supported PL/I programs, but it imposed restrictions on the PL/I program facilities available to a PL/I transaction program. This interface still exists and is described in many of the CICS manuals related to CICS macro-level coding, such as CICS/VS Application Programmer's Reference Manual (Macro Level). These manuals document PL/I-CICS/OS/VS restrictions, which are associated only with the PL/I-CICS interface that is supplied by OS CICS-Standard and CICS/OS/VS to PL/I-CICS users.

Since OS PL/I Release 3.1 and in subsequent releases, PL/I has supplied an interface between PL/I programs and the current release of CICS. This interface, like its CICS-supplied predecessor, consists of PL/I library modules (primarily from the PL/I Transient Library) modified for use in the CICS/OS/VS environment. It makes substantially more of the PL/I language usable in a CICS/OS/VS transaction program.

> **Note:** Many of the restrictions listed in CICS macro-level documentation as applying to PL/I apply only to PL/I programs using the old CICS-supplied interface, not to PL/I programs using the PL/I-supplied interface.

Each of the two PL/I-CICS interfaces supplies a module (DFHSAP) to be loaded as a part of the CICS/OS/VS nucleus and a module (DFHPL1OI) to be link-edited with your program. These modules

must match; that is, the CICS-supplied DFHSAP will not work with the PL/I-supplied version of DFHPL1OI and vice versa. A single execution of CICS/OS/VS can load only one DFHSAP; therefore, all PL/I-CICS transaction programs in a single execution of CICS/OS/VS (for instance, within a single region or address space) must use either the CICS-supplied interface or the PL/I-supplied interface, but no intermixing of the two is permitted. Under the current version of CICS/OS/VS, you might use one PL/I interface in one region or address space and the other PL/I interface in another, even though the CICS/OS/VS systems are not being executed independently of each other. If mixing should inadvertently occur, the results are as shown in Figure 155.

Your system programmer should ensure that the proper DFHSAP module is loaded with the CICS nucleus, and that the proper DFHPL1OI module is link-edited into transaction programs. It is sometimes helpful, however, to be able to tell which version of these modules is present. This can be done as follows:

- For DFHSAP, look at its link-edit listing (or a listing produced by AMBLIST) to see if it contains external names beginning with IBMF. If it does, it is the PL/I-supplied DFHSAP. If no such names are found, it is the CICS-supplied DFHSAP.
- For DFHPL1OI, look at its link-edit listing (or a listing produced by AMBLIST) to see what addresses are represented by entry-point names DFHPL1I, DFHPL1N, and DFHPL1C. If each points to a different location in DFHPL1OI, it is the PL/I-supplied DFHPL1OI. If they all point to the same location in DFHPL1OI, it is the CICS-supplied DFHPL1OI.

		CICS-Supplied	PL/I-Supplied
DFHSAP in CICS/VS Nucleus	CICS-Supplied	Supported - PL/I Function and restrictions as documented in CICS macro-level documentation.	Not Supported - Unpredictable CICS transaction abend, but probably ASRA for program check.
	PL/I-Supplied	Not supported - CICS ASRA (program check) transaction abend results.	Supported - PL/I function and restrictions as documented in this publication.

Figure 155. DFHPL1OI Link-Edited into Transaction

PL/I-CICS TRANSACTIONS

Because CICS supplies many facilities that would ordinarily be supplied by interactions between PL/I statements and the operating system, there must be ways of addressing CICS functional control blocks and requesting these services. Under the PL/I-supplied interface to CICS, some services (such as explicit allocation of BASED or CONTROLLED storage) are performed by the PL/I library using CICS/OS/VS facilities, but appear to you as the same ALLOCATE or FREE statements as would be used in a non-CICS program. Other services (such as I/O services similar to PL/I READ, WRITE, or REWRITE statements) are represented in the PL/I program as requests directed to CICS itself.

To implement these requests, CICS must define application program interface protocols. These protocols occur in two forms:

- Macro-level interface
- Command-level interface

Users are encouraged to use the CICS command-level interface, which is the newer and more flexible interface, for all new PL/I-CICS transactions. The command-level interface is required for all transactions that take advantage of the 31-bit addressing capabilities of MVS/XA.

MACRO-LEVEL INTERFACE

The macro-level interface has been supported by CICS since its earliest versions. It is invoked by including PL/I declarations for various CICS control blocks via %INCLUDE statements, coding user-supplied statements to access and alter these control blocks, and embedding CICS statements (in the form of assembler language macros) in the PL/I program. The program is then processed, in turn, by a CICS-supplied utility (called the CICS Preprocessor), the system assembler, and the PL/I compiler to produce an object module.

The effect of processing by the CICS preprocessor and the assembler is to convert the assembler macros into PL/I assignment statements that store values into CICS control blocks (in addition to any such statements already coded by the user), and a PL/I CALL DFHPL1I statement to convey the request to CICS/OS/VS. Incorrect addressing of CICS control blocks, erroneous or incomplete specification of requests, and use of incorrect data types cannot be diagnosed by PL/I at compile time, and, in many cases, cannot be diagnosed by PL/I or CICS at execution time. Such errors can cause application program errors, transaction abends, or even damage to CICS itself.

The detailed protocols for CICS macro-level coding can be found in CICS/VS Application Programmer's Reference Manual (Macro Level).

COMMAND-LEVEL INTERFACE

CICS/OS/VS has provided a set of programming protocols for CICS programming. This interface is invoked by coding statements in your application program and executing a CICS/OS/VS utility program called the CICS Translator. The statements are of this format:

```
EXEC CICS function [option[(arg)]]....;
```

The Translator supplies a control block (DFHEIB) for receipt of information from CICS/OS/VS, and a set of PL/I ENTRY declarations with parameter-list descriptors. It generates one PL/I CALL statement for each EXEC CICS command in the program. The program does not directly reference internal CICS control blocks and, in most cases, you do not need to address or manipulate such control blocks. All required parameters are present and of the correct data type for each CICS request, and request validation can be performed at execution time. This interface is called the command-level interface, or the High-Level Programming Interface (HLPI). It provides a simple and reliable way to code CICS transaction programs in PL/I.

Note: In the following special cases, the CICS command should be delineated by PL/I BEGIN and END statements:

1. When the CICS command is the on-unit portion of an ON statement. For example:

```
ON ERROR BEGIN;
           EXEC CICS DUMP;
         END;
```

2. When the CICS command is coded with one or more condition prefixes.

Using the PL/I-supplied interface permits either macro-level or command-level programs, or mixtures of the two, to be written in PL/I for CICS/OS/VS. It is strongly recommended that only command-level coding be used for new CICS/OS/VS-PL/I programming.

Although the command-level coding protocols permit extensive validation of EXEC CICS commands, neither PL/I nor CICS has any real way, under either set of CICS protocols, to diagnose use of the PL/I features listed as restrictions in Figure 154 on page 361. For example, the compiler would regard syntactically valid PL/I statements, such as READ, WRITE, or REWRITE, or calls to PLICKPT or PLISRTC, as perfectly valid, and would generate its usual object code for them. Execution of such restricted statements might have a serious impact on the integrity or performance of CICS/OS/VS, including termination of CICS itself, unpredictable transaction abends, system waits, and so on. Avoidance of restricted PL/I facilities in a CICS/OS/VS environment is your responsibility.

With the issues concerning macro-level versus command-level coding, and the CICS-supplied versus PL/I-supplied PL/I-CICS interfaces addressed by the above text and Figure 154 on page 361, the remainder of this chapter is devoted entirely to the PL/I-supplied interface.

COMPATIBILITY

Existing 24-bit addressing mode PL/I application programs (using CICS macro-level or command-level interface) can run with the PL/I Release 5.1 transient library on CICS Release 1.6.1 under MVS/XA without any changes. These programs can also be re-compiled and/or relink-edited with PL/I Release 5.1 compiler and libraries. However, PL/I application programs that use CICS macro-level coding must force AMODE 24 during link-editing with PL/I Release 5.1 libraries.

A PL/I application program already being executed using the PL/I interface from PL/I Release 3.1, or Release 4 and the CICS Command-Level interface stub from CICS Release 1.5.0, Release 1.6.0, or Release 1.6.1 will continue to execute on CICS Release 1.6.1 with PL/I Release 5.1 transient libraries. However, as soon as such a program is relink-edited with PL/I Release 5.1 CICS interface modules, it will no longer execute on CICS Release 1.6.0 or prior releases and it will no longer work with a PL/I release prior to Release 5.1.

Figure 156 on page 367 describes the valid combinations of different releases of PL/I with CICS/OS/VS Release 1.6.1.

Program check interruptions which raise the PL/I OVERFLOW, UNDERFLOW, FIXEDOVERFLOW, or ZERODIVIDE conditions will be handled with the PL/I Release 5.1 error handler under the STAE or NOSTAE options. These conditions were handled by PL/I Release 4.0 only under the STAE option.

Notes	PL/I Release 3.1 or 4.0			PL/I Release 5.1			CICS Release 1.6.1
	Compile	Link	Run	Compile	Link	Run	
1	*	*	*				With or without upgrade
2	*	*			*	*	With upgrade
2	*				*	*	With upgrade
3				*	*	*	With upgrade

Figure 156. Valid Combinations of PL/I Releases with CICS/OS/VS Release 1.6.

Notes to Figure 156

1 A program compiled, link edited, and run on PL/I Release 3.1 or 4.0 will continue to run as is on CICS Release 1.6.1, either with or without upgrade.

2 A program compiled and link-edited on PL/I Release 3.1 or 4.0, or a program compiled on Release 3.1 or 4.0 but relink-edited with the Release 5.1 resident library, can be run with the Release 5.1 PL/I transient library on CICS Release 1.6.1 with upgrade.

3 An existing program can be recompiled on Release 5.1, relink-edited with the PL/I Release 5.1 resident library, and run with the PL/I Release 5.1 transient library. New application programs will also be in this category. Execution of a PL/I Release 5.1 program requires CICS/OS/VS Release 1.6.1 with upgrade.

PL/I STORAGE

LIFETIME OF STORAGE ACQUIRED FROM CICS/OS/VS

When storage is acquired from CICS/OS/VS via a CICS GETMAIN request, that storage has a type (for example, USER, TERMINAL) that determines how CICS storage management will manage it. Storage acquired by a user directly from CICS/OS/VS via DFHSC TYPE=GETMAIN or EXEC CICS GETMAIN normally has a scope that spans the whole CICS task, not just the program. The storage remains allocated until it is freed, or until the CICS task ends. PL/I places storage acquired by the PL/I library, for either PL/I's Initial Storage Area (ISA) or a Secondary Storage Area (SSA), on a storage management queue associated with the current invocation of the program, not the task. When the program terminates, whether or not via PL/I termination, the program's PL/I storage will be freed, even though the task may still be active.

This distinction has major implications for storage passed back and forth between programs. Suppose, for a certain CICS transaction, PL/I program A links to PL/I program B, and a TWA or COMMAREA is available to hold a PL/I pointer to be communicated between the two. The TWA, since it is a part of the CICS task-related control block structure, remains available to both programs. CICS tries to ensure that a COMMAREA can be passed back and forth successfully, as described in the CICS/VS command-level coding documentation. Suppose, however, that the program tries to pass a pointer, via the TWA or COMMAREA, to some other storage area not in the TWA or COMMAREA. If B were to acquire the storage via a PL/I ALLOCATE statement, the storage would be released when B terminated, and thus could never be passed back to A. Any pointer in a TWA or COMMAREA

that pointed to such storage would be invalid, and the result of using it is unpredictable.

If A acquired the storage by issuing a PL/I ALLOCATE statement for a PL/I based variable, then A can convey the address of the storage to B, and B can use or alter the storage; however, B cannot free the storage. If B issued a PL/I FREE statement for the storage, PL/I storage management would not find it on its storage management chain for B. If B issued a CICS FREEMAIN, CICS would discover that it was PL/I storage, not user storage. Either of these requests would be in error.

If A acquired the storage by CICS GETMAIN, then A could convey the address of the storage to B and B could use, alter, or free the storage, since it would be user storage owned by the task, not by program A or B.

If the processing scenario called for B to acquire the storage and pass it back to A, then B would have to acquire the storage by CICS GETMAIN.

STORAGE CLASSES

The CICS user should avoid writing into STATIC storage, since changing STATIC storage violates reentrancy. Most or all user variables that are actually changed during program execution should be AUTOMATIC. User variables that have initial values, and whose values never change, should be declared STATIC INITIAL, and any variable declared EXTERNAL must have the INITIAL attribute to preclude generation of common CSECTs. Although AUTOMATIC storage provides reentrancy and should suffice for most purposes, you can allocate and free storage via ALLOCATE and FREE statements. BASED and CONTROLLED variables can be allocated and freed in this way.

CONTROLLED storage can be used on CICS/OS/VS. CONTROLLED variables are consistent with reentrancy. User references to based storage are handled via the pointer set by the ALLOCATE statement. The pointer can be AUTOMATIC.

The intent of CONTROLLED storage is to permit you to explicitly manage a push-down stack of multiple generations of variables. If you just want to explicitly allocate and free a piece of storage via PL/I ALLOCATE and FREE statements, BASED storage is more efficient than CONTROLLED storage.

"READ-ONLY" PL/I-CICS TRANSACTIONS

Under Release 5.1 of the PL/I Optimizing Compiler, the reentrant PL/I-CICS application programs are "Read-Only" and eligible to reside in the Link Pack Area (LPA) and Extended Link Pack Area (ELPA).

When PL/I application programs are placed in the LPA/ELPA, the overall performance will increase when they are shared by two or more CICS systems in the same processor. Integrity of these application programs and savings in the amount of storage use are other advantages of placing PL/I application programs in the LPA/ELPA.

See the CICS/OS/VS Version 1 Release 6 Modification 1 Installation and Operation Guide for information on placing a PL/I application program in the LPA/ELPA.

OUTPUT TO SYSPRINT

SYSPRINT can be used for any type of stream output. It is also used for error messages generated by the program and REPORT, FLOW, and COUNT output. Because CICS provides all normal I/O facilities, SYSPRINT is intended primarily for debugging. Performance may not be satisfactory for production programs. SYSPRINT is the only file that PL/I can write to. However, if another file is specified, the program may behave as if SYSPRINT had been specified.

SYSPRINT output is assigned to the CPLI transient data queue. The actual type of queue (Intra or Extra Partition) is determined during CICS installation. To learn the queue type in your installation, ask your system programmer.

Records sent to SYSPRINT take the form of the message, preceded by a terminal identification and a transaction identification. The whole record is preceded by an American National Standard control character to determine the format of the printing. The records are V-format with a maximum record length of 133. The lengths of the various fields are shown in Figure 157.

Because SYSPRINT output is transmitted to one queue from all transmitters, the queue may contain output from more than one PL/I program, and the records may be intermixed. Whether this occurs depends on how CICS is set up in your installation. In a debugging system that executes one transaction at a time, the queue contains output from only one PL/I program. In a system executing many transactions, output records from different PL/I programs are intermixed. If output records are intermixed, you must use an application program to sort the outputs of the various programs using the terminal and transaction identifiers as keys.

LL	00	ASA	terminal id	transaction id	output data
2	2	1	4	4	120

where LL is the length of the record, including the length bytes
00 is hex '00'
ASA is the American National Standard carriage control character

Note: LL00 part of the record is not printed.

Figure 157. Format of Records Sent to SYSPRINT

DECLARATION OF SYSPRINT

SYSPRINT need not be declared in your application program, but if it is, it should be declared as STREAM PRINT OUTPUT. Any ENVIRONMENT options that are specified are ignored. The PAGESIZE and LINESIZE option of OPEN can be used; all other options of OPEN are ignored. The maximum LINESIZE is 120; larger values are truncated.

SYSPRINT need not be explicitly opened or closed. However, if you are using CICS macro-level coding, it should be explicitly closed before the execution of any CICS facility that may result in control not returning to the PL/I program. For example, SYSPRINT should be explicitly closed before the use of a DFHPC macro with TYPE=XCTL, RETURN, or ABEND. If this is not done, the record being built at the time can not be transmitted.

The LINENO and COUNT built-in functions of PL/I stream I/O can be used against SYSPRINT, and return their proper values under OS. In the CICS/DOS/VS environment, however, they return zero. Thus, use of these built-in functions should be avoided if transaction portability between OS and DOS is to be maintained.

CHECK AND PUT DATA

Because of the extensive use of BASED storage in CICS/OS/VS transactions, you should remember the following restrictions on CHECK and PUT DATA.

In PL/I, it is not permissible to write:

```
PUT DATA (P -> VAR);
```

If VAR was declared as BASED (P), then the value of the generation of VAR to which P points can be written out by PUT DATA (VAR);.

CHECK cannot be raised for a based variable without a pointer specified in its declaration. In the case of VAR above, the value of VAR to which P points is supplied when CHECK is raised for VAR, even if some other pointer is used in the statement that raises CHECK. For example:

```
DCL P PTR,
    VAR BASED(P);
P -> VAR = 5;  /*prints VAR = 5 */
Q -> VAR = 8;  /*prints VAR = 5 */
```

No compile- or execution-time message will tell that the wrong generation of VAR is being printed out. CHECK must not be raised for variables in CICS control blocks used with the macro-level interface. If assigning to a variable in one of those control blocks raises CHECK for that variable, then in attempting to output the CHECK information by PL/I SYSPRINT transmitter, the value set by the user program may be destroyed. Since all the control blocks associated with command-level coding are read-only (from the user's point of view), CHECK can never be raised for them.

EXECUTION-TIME OPTIONS

Under CICS, execution-time options can be specified only in the PLIXOPT string. The PLIXOPT string is used in the manner described under "Specifying Execution-Time Options" on page 28. For example:

```
DCL PLIXOPT CHAR(20) VAR STATIC EXTERNAL
  INIT('ISASIZE(3000) NOSTAE');
```

The following options can be used. IBM recommended defaults are underlined.

```
FLOW | NOFLOW
COUNT | NOCOUNT
REPORT | NOREPORT
STAE | NOSTAE
ISASIZE(size)
ISAINC(size)
HEAP(size,increment,ANYWHERE|BELOW,KEEP|FREE)
```

FLOW, COUNT, and REPORT depend for their correct execution on PL/I termination being properly performed. See the discussion on "PL/I Program Termination" on page 377.

FLOW output is written on the SYSPRINT file whenever an on-unit with the SNAP option (for example, ON FINISH SNAP;) is executed. It is also included as part of PLIDUMP output if "T" is included in the dump option string.

CICS/OS/VS macro-level interface is incompatible with FLOW, COUNT, and REPORT (as well as with compiler options GONUMBER and GOSTMT). When an on-unit is entered and control is passed from a CICS/OS/VS macro, then linkage to FLOW, COUNT, REPORT, and statement-number tables is not available. This, with GONUMBER and GOSTMT, causes the statement identification in a message or PLIDUMP to be meaningless. With FLOW, COUNT, or REPORT, storage may be overlaid.

If an option is not specified, or if PLIXOPT is not specified, then the default options (except for FLOW and COUNT) are taken from the IBMBXOPT module generated during installation of the Transient Library. See your system programmer for these defaults. The default options for COUNT and FLOW are taken from the options specified at compile time (as for non-CICS systems). To avoid exposure to inappropriate non-CICS defaults for these options, it is a good practice to set all the options via a PLIXOPT string in each main procedure.

The SPIE option has no effect when used under CICS.

The STAE option specifies that PL/I error handling will be used for hardware-detected interrupts as well as CICS abends. See also, "Error Handling" on page 372.

Only the nonmultitasking arguments of ISASIZE apply and only positive values can be used. If too small a value is specified for ISASIZE, the minimum acceptable is acquired. This differs from OS practice. If the ISASIZE option is not specified, then an attempt is made to allocate an ISA sufficiently large to include both the standard control blocks and the initial allocation (DSA) for the main procedure. Such an attempt may fail if the FLOW option applies. This means that the minimum storage will be acquired, and any storage obtained other than that for static and the main procedure's variables will have to be requested from CICS. This may result in slow execution.

If too large a value is specified for ISASIZE, the allocation may fail, causing CICS to terminate the program.

To determine the optimum ISASIZE, you should use the REPORT option. The fastest initialization will be achieved if a positive ISASIZE is specified that is large enough to hold the storage requirements of the first block. The fastest execution will be achieved if all PL/I storage can be obtained from the ISA, and this is the meaningful goal in most cases. For further details, see "Execution-Time Storage Requirements for Nonmultitasking Programs" on page 37.

If ISAINC is specified, for storage requests not satisfied within ISA, the larger of ISAINC size or the requested size will be acquired as increments. The maximum ISAINC size is 65,496 and will be rounded up to the next multiple of 8 bytes.

Appropriate values for ISASIZE and ISAINC can significantly reduce the number of CICS GETMAINs and FREEMAINs required for execution of your PL/I program.

When HEAP is specified, a separate heap area is utilized for CONTROLLED and dynamically allocated BASED variables. The maximum HEAP size and increment if specified is 65,496 for below and 1,073,741,816 for above the 16M line. Heap size and increments will be rounded up to the next multiple of 8 bytes.

Provided the STAE option is in effect, PL/I error handling is the same as under OS. The only exception is that it is possible, under CICS, to override the generation of an error message when the ERROR condition arises. This can be useful in a production program where the transmission of a message to the CPLI queue may be an inappropriate reaction to an error.

The error message is suppressed if an on-unit for the ERROR condition is supplied. If you require both the on-unit and the message, you should specify SNAP in the on-unit. For example:

```
ON ERROR SNAP
   BEGIN;
      ON ERROR SYSTEM;
      PUT DATA (A,B,C);
      EXEC CICS DUMP ... ;
      CALL PLIDUMP (...);
   END;
```

All error messages are transmitted to the SYSPRINT file, which, as described above, is attached to the CPLI queue.

If the NOSTAE option is in effect, program check interruptions which raise the PL/I OVERFLOW, UNDERFLOW, FIXEDOVERFLOW, or ZERODIVIDE conditons will be handled with the PL/I Release 5.1 error handler. CICS abends are handled by CICS. The default CICS action is to produce a dump and terminate the transaction.

STAE allows PL/I interrupts that arise from either hardware interrupts or CICS/OS/VS transaction abends to be handled by the user in on-units; otherwise, such errors cause a CICS task abend. Software-detected PL/I interrupts (for example, CONVERSION, or ERROR because a negative argument was supplied to the real square root function) cause PL/I conditions to be raised whether or not STAE is in effect. Software-detected PL/I conditions can be raised even if NOSTAE is in effect.

PL/I does not issue SPIE/ESPIE or STAE/ESTAE macros in the CICS environment. If the STAE option is requested, an EXEC CICS HANDLE ABEND command is issued by PL/I initialization. If a program check occurs, the CICS SPIE routine (DFHSRP) gets control and, if the program check occurred in user code in a PL/I transaction program, invokes the PL/I error handler. Thus, the STAE option does not affect CICS/OS/VS itself or any other CICS transaction. It uses CICS/OS/VS error handling; it does not override it.

If STAE is specified, CICS/OS/VS control program services address the CICS version of the PL/I error handler as an exit routine to CICS/OS/VS control program services. As such an exit routine, the PL/I error handler handles any CICS/OS/VS abends (whether initiated by program checks or by software elsewhere in CICS) that occur in the PL/I program or associated CICS services.

Use of the DFHPC TYPE=SETXIT macro or EXEC CICS HANDLE ABEND, while the STAE option is in effect, will remove the PL/I error handling facilities; that is, the effect is as if NOSTAE were specified. However, interrupts may result in CICS receiving control with an incorrect program mask that could lead to unexpected program check interrupts in other transactions. If you wish to use CICS facilities to set your own error exit, you should use the NOSTAE option. Use of the STAE option results in PL/I specifying its own error exit, and the respecifying of such an exit leads to unpredictable results.

PL/I error-handling facilities function in a way that is compatible with CICS's own error-handling facilities. For example, CICS/OS/VS Dynamic Transaction Backout may be needed to back out updates already done by a transaction that has failed, even though the error may have been detected internally within the program, not by CICS/OS/VS (for example, a PL/I software

interrupt raised ERROR), or a CICS-initiated transaction abend was temporarily intercepted but not successfully handled by a PL/I on-unit. Furthermore, if program A links to program B, and B abends, A must be able to obtain that information and make use of it.

To meet these requirements, the PL/I error handler under CICS/OS/VS does several things:

- If STAE is in effect so that the PL/I error handler gets control after a CICS-initiated abend, the on-units (if present) in the user program may or may not successfully effect recovery from the error condition. If they do not, the ultimate effect in the PL/I program is to raise ERROR. If there is no ERROR on-unit, or if the program takes normal return from the ERROR on-unit, then PL/I termination issues a CICS abend using the original CICS abend code (or using APLS if the original code was ASRA). Thus, the temporary but ineffectual interception of the CICS abend will not keep the transaction from abending, and will not keep Dynamic Transaction Backout (for example) from functioning. If code in PL/I on-units successfully recovers from the problem, the transaction continues and no abend occurs.

- Whether or not STAE is in effect, PL/I software interrupts as well as hardware interrupts such as OVERFLOW, UNDERFLOW, FIXEDOVERFLOW, and ZERODIVIDE interrupts can occur and cause appropriate PL/I conditions to be raised. If not corrected in appropriate on-units, the software interrupt eventually causes ERROR to be raised. If there is no ERROR on-unit, or if the program takes normal return from the ERROR on-unit, then PL/I termination communicates to CICS/OS/VS this termination-in-error of the transaction by issuing a CICS/OS/VS abend with abend code APLS. Thus, Dynamic Transaction Backout (for example) can proceed just as though CICS/OS/VS had initiated the abend.

- When program A links to program B, and program B abends, then upon completion of CICS/OS/VS termination of B (possibly after some attempt at error recovery in B has been unsuccessful), CICS initiates the abend of A (as the program that linked to B). If A is a PL/I program being executed with the STAE option, then ERROR will be raised in A with condition code 9050, meaning "An abend has occurred." If A has some way of making the transaction continue, it may do so by exiting from the ERROR on-unit via a GO TO statement rather than by normal return.

The support for PL/I error handling makes it quite possible for you to cope with computational interrupts, CONVERSION errors, and other non-I/O-related conditions using the same PL/I facilities that would be used in programs executed directly under OS.

For conditions associated with CICS/OS/VS abends (including ASRA abends for program checks), CICS itself provides a facility (DFHPC TYPE=SETXIT or EXEC CICS HANDLE ABEND;). This facility allows branching to either a program external to the currently executing program or to a routine located within the current program. PL/I issues an EXEC CICS HANDLE ABEND that identifies the PL/I error handler as a routine to establish linkage from CICS to the PL/I error handler, so your use of this facility will necessarily destroy PL/I error handling. In PL/I programs, CICS/OS/VS does not support a SETXIT identifying the label of a routine in the current program. This is not really a restriction, since PL/I ON, SIGNAL, and REVERT statements give you all the facilities of PL/I to do so anyway.

PL/I error-handling facilities do not include I/O-related conditions like RECORD, TRANSMIT, ENDFILE, KEY, and so on, because I/O is not performed using PL/I files and PL/I I/O statements, but by CICS file-handling facilities. (SYSPRINT is the sole exception to this rule.) Conditions detected by CICS/OS/VS during the processing of your macro or command are

reflected to you via CICS-defined protocols. These are described in the CICS manuals.

In command-level programs, such conditions are reflected to you based on previously executed EXEC CICS HANDLE statements.

The EXEC CICS HANDLE facility semantically resembles a PL/I on-unit with the syntax:

```
ON condition GO TO label;
```

The HANDLE command can be coded wherever you could code the ON... GO TO ... statement. The label to be branched to can be located in some other active block and the condition can arise in some still later block. HANDLE will terminate intervening PL/I blocks by invoking PL/I's out-of-block GO TO facilities.

HANDLE is not semantically identical to the "ON condition GO TO label;" statement. A PL/I on-unit disappears when the block containing it terminates; a CICS HANDLE disappears when it is explicitly overridden by another one. Thus a HANDLE command could specify a branch to a label in a block no longer active. Since HANDLE is implemented by forcing a PL/I out-of-block GO TO, this is like assigning a label constant to a PL/I label variable and then branching to the label variable after the block containing the label constant has terminated. This is an invalid GO TO. The PL/I out-of-block GO TO mechanism attempts to detect this error and raises ERROR when it detects it. If PL/I out-of-block GO TO fails to detect such an invalid GO TO, however, the GO TO becomes a wild branch that will cause some unpredictable failure. Thus, upon return from a PL/I block that established HANDLE for some particular condition, your program should issue a resetting HANDLE for that condition (provided, of course, that there is still some possibility of the condition arising). This resetting is unnecessary for a PL/I on-unit.

ABEND CODES USED BY PL/I UNDER CICS

Certain error conditions result in the PL/I library routines issuing CICS abends. Such abends are not caught by the PL/I error handling facilities, even if the STAE option is in effect, since the PL/I abend exit is cancelled. They will, therefore, normally terminate the transaction and produce a dump. Abend codes used are:

APLC The shared library facilities are required by the application program, but were not included in the CICS system during initialization/installation. See your system programmer.

APLE An error occurred during PL/I program management (equivalent to a 4000 abend on non-CICS systems).

APLI An error was detected by CICS on transmission of a record to the CPLI queue. See your system programmer.

APLM No main procedure.

APLD An error was detected by CICS on transmission of a record to the CPLD queue. See your system programmer.

APLG A get storage request to the storage allocation routine specified a size greater than the CICS/OS/VS permitted maximum of 65,496 (or a maximum of 1,073,741,816 under MVS/XA). This error is caused by having either a based or controlled variable that is too large in an ALLOCATE statement, or too many large AUTOMATIC variables.

APLS This abend is issued on termination, if termination is caused by the ERROR condition, and the ERROR condition was not caused by an abend (other than an ASRA abend).

This is the abend code issued by PL/I when a transaction terminates in error due to a PL/I software interrupt (CONVERSION, for example), and there is no ERROR on-unit, or the program takes normal return from the ERROR on-unit. Since the program failed, the failure must be reflected to CICS/OS/VS as an abend so that Dynamic Transaction Backout, and so on, can occur if necessary. Since there was no CICS/OS/VS abend to be reissued, PL/I termination must supply an abend code.

APLS is also the abend code issued by PL/I termination when a program check (CICS ASRA abend) is intercepted by the PL/I error handler, but the condition cannot be resolved by the user. For instance, the program was terminated due to normal return from an on-unit. PL/I cannot re-issue the abend with code ASRA, because a program that linked to this failing program would be abended with an abend code (ASRA) that implies that a PSW and registers are being supplied that permit some sort of fixup or retry; in fact, the PSW is that of a program that is no longer active, and the registers point to storage locations that are no longer meaningful. For more information on ASRA, refer to CICS Message and Codes.

APLX The total possible LIFO storage segments have been exhausted. Check the program for loops or increase the ISASIZE or ISAINC.

XXXX An abend is issued with the original abend code if an abend other than ASRA caused the ERROR condition to be raised and this caused termination, and no IBMBEER module was included to cause user-specified action for the ERROR condition.

IBMBEERA

When IMBEERA is included in the DFHSAP module, the abend facility will be available under CICS.

IBMBEER's return code indicates whether a simple return or an abend is to be issued. IBMBEER is described in OS PL/I Optimizing Compiler: Installation Guide for MVS.

USE OF PLIDUMP

The CALL PLIDUMP statement is used to obtain a dump of PL/I storage areas in PL/I terms. Areas to be dumped can be specified via an options list in the same way as on non-CICS systems. Most of the code involved is dynamically loaded, so the resident storage requirements are small, although a larger amount of storage is required when the statement is executed. This means that CALL PLIDUMP statements may be included in production programs to be executed if unexpected errors arise.

The following options are available:

T Trace of active procedures, etc.
NT No trace
S Terminate execution
C Continue execution
B Produce a hexadecimal dump of PL/I control blocks (DSAs, PL/I TCA, etc.) NOTE: This option is effective only with "T" option.
NB No dump of PL/I blocks
K Produce a hexadecimal dump of the CICS Transaction Work Area (TWA)
NK No dump of CICS TWA block.

The default values are T, C, NB, and NK.

The dump information is built up into records, suitable for printing, that are transmitted to a transient data queue with a destination ID of CPLD.

Each record consists of a 1-byte American National Standard control character followed by up to 120 bytes of data. The first record transmitted by a CALL PLIDUMP statement is an identification record that contains the terminal ID, transaction ID, transaction number, date, and time.

Prior to transmitting this record, an ENQ is issued. The corresponding DEQ is issued after the last record to be produced by the CALL PLIDUMP statement has been transmitted. This means that, at any one time, no more than one transaction will be producing a PLIDUMP, and that all the records for each PLIDUMP are together on the queue. Therefore, this queue can be sent directly to a printer, if desired. For details about how a dump is printed, contact your system programmer.

Since PLIDUMP does not print the program or its static storage, and since there are many CICS/OS/VS control blocks that it does not print, it may be appropriate to request a CICS dump in addition to PLIDUMP.

Under CICS/OS/VS, output from the FLOW, COUNT, or REPORT options goes to SYSPRINT, not PLIDUMP.

PLIDUMP copes with program checks that arise during its own execution; however, it is unable to cope with program checks in the CICS/OS/VS environment unless the program being dumped was being executed with the STAE option in effect.

INTERLANGUAGE COMMUNICATION—OPTIONS ASSEMBLER

The OPTIONS attribute with ASSEMBLER can be used under CICS, allowing assembler language subroutines to be called from a PL/I routine and the arguments passed in an assembler language manner. See OS and DOS PL/I Language Reference Manual for details. No other interlanguage communication is allowed.

If CICS facilities are requested from a macro-level assembler language program, the registers must be set to the CICS conventions before the facility is used, and reset to PL/I conventions afterward. For this reason, it is inadvisable to use CICS facilities from a macro-level assembler language subroutine. Similarly, it is inadvisable to call any other system facilities. Macro-level assembler language routines should only be used for computational purposes.

See CICS/VS documentation for use of CICS/OS/VS command-level facilities in an assembler language subroutine.

STORAGE AND CURRENTSTORAGE

The STORAGE and CURRENTSTORAGE built-in functions return the length of an item to your PL/I program. This is useful with CICS, where functions often require the length of an argument as well as its address. In particular, these functions can be used with the command-level interface to get lengths of PL/I aggregates without your having to count or compute such lengths or specify length fields in the CICS commands.

PL/I PROGRAM TERMINATION

From your point of view, most PL/I programs terminate by simply returning from your main procedure, and you may even regard this as a return to the operating system. In fact, it is a return to PL/I initialization/termination to perform various cleanup functions.

If errors occur during program execution, the ERROR condition may be raised. If there is no ERROR on-unit (or if there is an ERROR on-unit and control exits via normal return, not via a GOTO statement), the PL/I program will terminate, via PL/I termination facilities. A small percentage of PL/I programs terminate via a STOP statement or a SIGNAL FINISH statement (although SIGNAL FINISH is not an operative statement if a FINISH on-unit (even a null one) has not been established). All of these, however, cause the program to terminate via PL/I termination facilities.

In the CICS/OS/VS environment, PL/I programs may terminate in any of the above ways, or they may terminate via CICS/OS/VS statements.

Using command-level coding, the commands EXEC CICS RETURN, EXEC CICS XCTL, or EXEC CICS ABEND terminate the PL/I program via PL/I termination facilities, because the CICS EXEC Interface Program (DFHEIP) branches into the PL/I termination routine to ensure that PL/I termination processing occurs.

Using the CICS macro-level coding interface, however, the macros DFHPC TYPE=RETURN, DFHPC TYPE=ABEND, and DFHPC TYPE=XCTL cause branches directly into CICS Program Control Program (DFHPCP), terminating the PL/I program without executing PL/I termination code. Thus, nothing dependent on PL/I termination processing can work. This means that, in a macro-level program terminated by the above DFHPC macros:

- SYSPRINT output is lost unless you insert a CLOSE statement for SYSPRINT.
- Output from the FLOW, COUNT, and REPORT options is lost.

For straightforward termination, the DFHPC TYPE=RETURN macro can usually be changed to a PL/I RETURN or SIGNAL FINISH statement, reinstating normal PL/I termination. There is no comparable way to convert DFHPC TYPE=XCTL, except to approximate it by DFHPC TYPE=LINK followed by a PL/I RETURN or SIGNAL FINISH to end the PL/I program. This may be an undesirable circumvention from a CICS point of view. The long-range solution is to convert the program to use the command-level interface.

When a PL/I program terminates via normal PL/I facilities or CICS commands, the following occurs:

1. Any requested FLOW, COUNT, or REPORT output is written to SYSPRINT.
2. SYSPRINT is closed (if it is open).
3. All storage acquired by the PL/I libraries is freed before control is returned to CICS.

PL/I SHARED LIBRARY FOR CICS/OS/VS

The Shared Library facility is available to CICS/OS/VS PL/I Optimizing Compiler users. When the PL/I Shared Library is installed, all the PL/I programs must be re-link-edited to include the PLISHRE module.

CICS/OS/VS initialization issues system LOAD macros to load the two shared library load modules, IBMBPSLA and IBMBPSMA. Their addresses are saved by CICS/OS/VS. Thereafter, whenever PL/I initialization (IBMFPIRA) initializes a PL/I program, it checks

the PLISTART parameter list in DFHPL1OI to see whether IBMBPSRA is resolved, that is, if the user included PLISHRE. If the answer is "yes," it looks to see if the shared library modules have been loaded by CICS/OS/VS.

If the shared library modules have not been loaded, the transaction is abended since it cannot be executed. If they have been, then their addresses are moved into the slots in the PL/I TCA where the code in the link-edited shared library bootstrap module expects to find them. Thereafter, the shared library is used by the CICS/OS/VS transaction just as it can be used by any PL/I program executed directly under OS. The idea, normally, is that the shared library modules are in the link pack area; if, however, no PL/I modules were being used except under CICS/OS/VS, the shared library could be loaded into the CICS/OS/VS address space or region.

Use of the PL/I shared library results in a smaller composite load module and may increase performance. This is especially important in CICS environments with virtual storage constraints and slow performance. For information on how to create a shared library which most suits your requirements, see the OS PL/I Optimizing Compiler: Installation Guide for MVS.

CICS/OS/VS SYSGEN provides a step related to a data set called DFHSHRE and the PL/I Shared Library. A function commonly placed in the shared library is PL/I Initialization/Termination, and in this case the PL/I initialization/termination entry points are resolved by PLISHRE. In a CICS environment, they must be resolved in DFHPL1OI. Therefore the CICS/OS/VS SYSGEN creates a private CICS/OS/VS version of PLISHRE in which those entry points have been disabled. CICS/VS transactions are linked with the CICS/OS/VS version of PLISHRE in DFHSHRE. Non-CICS/OS/VS PL/I programs are linked with the unmodified PLISHRE.

LINK-EDITING PL/I-CICS APPLICATIONS

PL/I-CICS application programs must be link-edited in a different way than non-CICS applications. This is because the normal entry point, control section PLISTART, is not required on CICS systems. Instead, the module DFHPL1OI is provided, which acts as the entry point to the program and must be link-edited with the application program. The CICS loader requires that DFHPL1OI be positioned at the head of the load module.

See CICS/OS/VS Installation and Operation Guide for information about installing your PL/I application programs.

Assuming the PL/I application is made up of members PLIAPP and EXTSUB, which reside in the library defined by ddname L1, and that the SYSLIB data set contains DFHPL1OI, then the following linkage-editor statements should be used:

```
INCLUDE  SYSLIB (DFHPL1OI)
      Ensure that DFHPL1OI is at the
      head of the load module
REPLACE  PLISTART
      Delete unwanted CSECTs from
      following INCLUDE
INCLUDE  L1 (PLIAPP)
      Application procedure (1)
REPLACE  PLISTART
      Delete unwanted CSECTs from
      following INCLUDE
INCLUDE  L1 (EXTSUB)
      Application procedure (2)
INCLUDE  DFHSHRE (PLISHRE)
      Optional. Use if shared library
      is to be used.
NAME     APROG (R)
      Optional. Defines name of
      resident load module.
```

If the PL/I application uses IMS while running under CICS, add the following statements after the REPLACE PLISTART statements above:

- For interface via CALL PLITDLI, add

  REPLACE IBMBDLIA (PLITDLI)

- For interface via CALL ASMTDLI, add

  REPLACE IBMBDLIB (ASMTDLI)

- For interface via EXEC DLI, add

  REPLACE IBMBDLIC (DFHEI01)

If the PL/I application uses EXEC CICS commands, add the following statement after the REPLACE PLISTART statements above:

REPLACE IBMBDLIC (DFHEI01)

PL/I-CICS/OS/VS INTERFACE COMPONENTS

PL/I supplies a program interface module called DFHPL1OI and a nucleus interface module called DFHSAP. DFHSAP is a part of the PL/I product (supplied in the Transient Library), not of CICS/OS/VS, but it is loaded during CICS/OS/VS initialization to become part of the CICS nucleus.

DFHSAP's initialization module establishes PL/I execution options for each CICS-PL/I program. Its PL/I error handler is a proper PL/I error handler. It contains modified versions of various OS PL/I modules.

Certain other functions, normally required only in a debugging environment, are implemented by loading PL/I transients into CICS/OS/VS storage via an EXEC CICS LOAD command. Such transients include the STREAM OUTPUT PRINT transmitter for SYSPRINT, the PLIDUMP transients, the storage management module required for the REPORT option, and two versions of the PL/I execution-time messages module (one version for GONUMBER/GOSTMT, the other for NOGONUMBER/NOGOSTMT). Just as all the modules in DFHSAP are tailored for CICS, so these PL/I-CICS/OS/VS transients are all CICS-tailored modules, although they are very similar to their OS PL/I Transient Library counterparts.

There is no compile-time CICS option; however, PL/I library modules can tell whether or not they are being executed in the CICS environment by testing a bit, TTKK, in the PL/I TCA, and some of them make use of a CICS implementation appendage, built in the PL/I Program Management Area right after the PL/I TCA and TIA. It is principally used by the modules in DFHSAP, but various other library modules have sections of code for the CICS environment that test this bit and use the CICS appendage.

The CICS appendage is described in OS PL/I Optimizing Compiler: Execution Logic Manual.

PL/I-CICS/OS/VS APPLICATION PROGRAM INTERFACE (DFHPL1OI)

The link-edited CICS/OS/VS interface module (DFHPL1OI) replaces the batch-mode PLISTART CSECT, and contains what amounts to a CICS/OS/VS-tailored version of the PLISTART parameter list.

Execution of your PL/I transaction program commences when DFHPCP calls DFHPL1OI at its entry point DFHPL1N. DFHPL1N immediately calls PL/I initialization in DFHSAP, passing it the following:

The addresses of:

PLIMAIN Address of MAIN procedure.

PLIFLOW Flow trace initialization module (if FLOW option).

PLICOUNT Count initialization module (if COUNT option).

PLIXHD User's heading for COUNT and REPORT output.

PLITCIC Entry to CICS/OS/VS HLPI.

IBMBPSRA Shared library transfer vector.

IBMBPOPT Compiler-parsed PLIXOPT options.

IBMBERCA CHECK module.

plus the length of pseudo-register vector (PRV).

Any of the above parameter addresses except PLIMAIN can be zero if the related entity or option does not exist for the program being initialized.

In addition to the code at entry point DFHPL1N to pass the above parameter list to PL/I initialization in DFHSAP, DFHPL1OI also contains the following functional entry points:

DFHPL1I Bootstrap to CICS services for the macro-level interface

DFHPL1C Entry point to return CSA address to caller.

IBMBOCLA
IBMBOCLB
IBMBOCLC Entry points to branch through DFHSAP to OPEN/CLOSE code in STREAM PRINT transmitter for SYSPRINT.

IBMBKDMA Bootstrap to PLIDUMP

It also provides LOAD and RELEASE services for the resident libraries.

PL/I CICS/OS/VS NUCLEUS INTERFACE MODULE (DFHSAP)

The PL/I interface module in the CICS/OS/VS nucleus is a PL/I-provided module that supports PL/I Optimizing Compiler programs only. It consists of bootstrap code plus OS PL/I library modules modified for the CICS/OS/VS environment. These modules have names that begin IBMF instead of IBMB in batch.

DFHSAP contains the following library modules:

IBMFPCCA
Bootstrap to library modules in DFHSAP.

IBMBOCLA, B, C
IBMBSTVA, B, C
Bootstrap code to invoke (via CICS LOAD the first time) the STREAM OUTPUT PRINT transmitter for SYSPRINT (plus OPEN/CLOSE).

IBMFPIRA
PL/I initialization/termination code.

IBMFPGRA
Storage management without REPORT option.

IBMFERRA
PL/I error handler.

IBMBXOPT
Default execution-time options CSECT created when user installed OS PL/I Transient Library.

Note the presence of IBMBXOPT. If an execution-time option is not set via PLIXOPT, it is set to its batch mode default as specified during installation of the PL/I Transient Library. These batch default options may not be well-suited to the CICS environment. It is good coding practice to set all of these options explicitly in a PLIXOPT string.

The modules in DFHSAP supply all the initialization/termination, storage management, and error-handling function required in a debugged production program. In the testing and tuning environment, however, the SYSPRINT facility, PLIDUMP, FLOW, COUNT, REPORT, and CHECK may be desired, and error messages concerning failing programs will likely be produced on SYSPRINT. These functions require transients that are part of the PL/I Transient Library, but are tailored for the CICS environment and are loaded into CICS storage by an EXEC CICS LOAD command. CICS/OS/OS regards them as ordinary transaction programs, and macros for their PPT entries are on the PL/I distribution tape. They are:

IBMFSTVA
STREAM OUTPUT PRINT transmitter, altered to handle only the CICS/OS/VS version of SYSPRINT, but with OPEN/CLOSE support added.

IBMFPGDA
Storage management with REPORT option.

IBMFPMRA
Module to generate storage report for REPORT option.

IBMFEFCA
Module to produce COUNT output.

IBMFESMA, IBMFESNA, IBMBEOCA, IBMBETxA
Messages modules.

IBMFKMRA, KPTA, KTCA, KTRA, KTBA, KCSA
PLIDUMP modules.

Two PL/I Resident Library modules test the CICS bit in the PL/I TCA and take slightly different paths based on it. However, they do not interface with CICS directly. They are:

IBMBSIOA
The stream initialization output module that, while building a PL/I block called the SIOCB, has to get the address of the PL/I File Control Block (FCB). This address is obtained differently for the CICS SYSPRINT file than for ordinary batch STREAM OUTPUT files. Subsequent stream I/O modules address the file via the SIOCB and thus require no modification to run in the CICS environment.

IBMBCCSA
The complex string director module uses the load service in DFHPL1OI to load the necessary modules under CICS.

APPENDIX A. VSAM BACKGROUND

This appendix gives an introduction to the facilities of VSAM and Access Method Services. The commands for creating and deleting data sets, and for creating alternate indexes, are described. Other housekeeping tasks are described in your Access Method Services manual. If you have complex requirements or are going to be a frequent user of VSAM, you should review the VSAM publications for your operating system.

PL/I does not support all VSAM function.

THE VSAM CATALOG

VSAM data sets must be defined and cataloged in a VSAM catalog before they are loaded with data. Each VSAM data set's name and physical attributes are recorded in the catalog. A hierarchy of catalogs is possible, in which you have your own private catalog, which in turn is cataloged in the master catalog. Alternatively, you may catalog your data sets directly in the master catalog.

Data sets are defined and cataloged by using the Access Method Services program.

By having all data sets cataloged, close control of your data sets is possible and JCL can be restricted to simply associating the name of the data set with the file name in the PL/I program. Any other information necessary to use the data set will be found in the catalog. Thus, when using VSAM, essential JCL is reduced to the use of the DSNAME parameter and the DISP parameter. Other information can be supplied but it is merely used to override defaults and tailor VSAM's processing to suit your needs in matters such as buffer size.

VSAM DATA SETS

The three types of VSAM data sets are:

- A key-sequenced data set, which consists of a data component containing records with embedded keys, and an index component relating key values to relative locations of the records. The index, created and maintained by VSAM when data is written, is called the prime index.

  You may retrieve records directly, by supplying a key value as a search argument, or sequentially. Records retrieved sequentially are returned in order of their key values, and not their location in the data set.

  To create a key-sequenced data set, records must be presented in order of key values. Once a key-sequenced data set has been created, VSAM permits a full range of operations upon the data — retrieval, insertion, deletion, and changing the length of a record — with either sequential or direct-access.

  For a key-sequenced data set, VSAM also permits access to control intervals and access by relative byte address; however, PL/I does not support these types of access.

- An entry-sequenced data set, in which the records are in the order in which they were presented for storage (that is, each new record is stored at the end). Once you have created an entry-sequenced data set, records cannot be inserted, deleted, shortened, lengthened, or moved from one location to another. They may, however, be replaced with records of the same data length.

An entry-sequenced data set is essentially a sequential data set, but one whose records can be updated and can be retrieved either sequentially or at random by direct-access. The search argument for direct retrieval is a record's relative byte address (RBA), that is, its displacement from the start of the data set. To retrieve records randomly, your program must keep track of records' RBAs and associate RBAs with the contents of records. VSAM makes the RBA available after each record is written.

- A relative record data set, which is a string of fixed-length record slots, each of which is identified by a relative-record number from 1 to n, where n is the maximum number of records that can be stored in the data set. Each record occupies a single slot and is stored and retrieved by an argument which is the relative-record number of the slot. The size of each slot is the record length you specified when you defined the data set.

All VSAM data sets must be on direct-access storage devices. Under VSAM it is therefore possible to access records in all types of data sets by means of a key.

VSAM's use of catalogs to hold information about the physical attributes of all data sets, and the use of a separate service program (Access Method Services) for data set management, results in a reduced dependence on JCL compared with other access methods. It has the advantage that operations on data sets are more explicitly specified using VSAM. This has the corresponding disadvantage that temporary data sets cannot be so easily created for the length of the execution of a program. To compensate for this, the REUSE option of the DEFINE CLUSTER command specifies data sets that are to be used as temporary work areas. REUSE is further described later in this appendix.

The physical organization of VSAM data sets differs considerably from those used by other access methods. VSAM does not use the concept of blocking, and, except for relative record data sets, records need never be of a fixed length. VSAM data sets are held in control intervals and control areas; the size of these is normally determined by the access method and the way in which they are used is not visible to you. Consequently, concern about blocking factors and record length is largely removed by VSAM although records cannot, of course, exceed the maximum specified size.

ACCESS METHOD SERVICES

Access Method Services is a multifunction service program that carries out utility tasks on VSAM data sets. It is used to define them (that is, to record them in a catalog), to delete them, to generate alternate indexes from them, and to carry out many other routine tasks. You request tasks that you want by coding the appropriate Access Method Services commands and executing the Access Method Services program.

Access Method Services may be used in a separate job or job step in a batch system, called from a PL/I user program, or specified by command in an interactive system. In a batch system, the EXEC statement

```
//STEP  EXEC  PGM=IDCAMS
```

is used and the commands placed in the file SYSIN. On TSO, you enter the commands as if they were TSO commands. On CMS, you include the commands in a file with the filetype AMSERV, and specify the name of the file in the AMSERV command.

To create a data set you use the DEFINE CLUSTER command of Access Method Services. A cluster can be a key-sequenced data set, which consists of a data component and an index component, or it can be an entry-sequenced or relative record data set, which consists of only a data component. The command specifies

the name to be used for the data set, the amount of space required, the volume on which it will be placed, the record length, the position of any key, the catalog in which it will be recorded, and, optionally, a number of other physical attributes. For example:

```
DEFINE CLUSTER (NAME (BLOGGS) -
  VOL(HUR137) CYL(1 1) -
   RECSIZE(20 80) KEYS(10 0))
```

This defines a key sequence data set called BLOGGS on the volume HUR137. One cylinder is to be allocated as a primary space allocation and secondary allocations are to be in increments of one cylinder. The record size varies with a maximum of 80 bytes and an average of 20. The key is 10 bytes long and starts in the first byte (offset 0).

PASSWORD PROTECTION

VSAM data sets can have password protection, allowing access to be limited to those who know the password. Various levels of password can be provided to give different degrees of access to the data set.

The master password allows complete access to read, write, and delete the data set. Access to alter the contents of the data set but not to delete it is given in the update password. Access to read the data set, but not to alter it, is given in the read password. These three are the only levels of password that concern you as a PL/I user. However, there is a fourth level between the master password and the update password that allows the data set to be accessed at the control interval level, but does not allow the data set to be deleted; this is the control password. PL/I does not support control interval processing.

Passwords are set when the data set is defined using Access Method Services, and can be altered using the ALTER command of Access Method Services. For a data set to be protected, it is necessary for the catalog that contains it, and the master catalog, to be protected.

THE LIFE OF A VSAM DATA SET

A VSAM data set passes through four stages during its life:

1. Definition with the DEFINE command of Access Method Services.
2. Initial loading. Before a newly-defined key-sequenced data set is used for UPDATE or INPUT, it must be loaded by writing the initial data. This can be done from a PL/I program. After this point an alternate index may be defined and a path built, using Access Method Services.
3. Updating and reading, when the data is read from the data set or the original data is altered. Again this can be done from the PL/I program.
4. Deletion with the DELETE command of Access Method Services.

DEFINING A VSAM DATA SET

VSAM data sets are defined and cataloged using the DEFINE CLUSTER command of Access Method Services. To use the DEFINE command, you need to know:

- If the master catalog is password protected, the name and password of the master catalog or 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.

If the space is not available for your data set, you must use the DEFINE command to define space before you define your data set. The method of defining space is fully explained in the Access Method Services manual. Your system programmer will be able to tell you if space has been defined.

DEFINE CLUSTER COMMAND

The DEFINE CLUSTER command is the command that defines and catalogs your data set. The simplified syntax of the command is:

Syntax

```
DEFINE CLUSTER (NAME(data-set-name) -
  VOLUMES(volser1 [volsern]) -
  [FILE(ddname)] -
  [REUSE|NOREUSE] -
  INDEXED|NONINDEXED|NUMBERED -
  [KEYS(length offset)] -
  [FREESPACE(ci% ca%)] -
  TRACKS|CYLINDERS|RECORDS -
   (primary-alloc secondary-alloc) -
  RECORDSIZE(average maximum) -
  password-options -
  [SHAREOPTIONS(n[ m])] -
  other-options) -
[DATA (options)] -
[INDEX (options)] -
[CATALOG (catname/password)]
```

Items in uppercase (capital letters) must be coded as shown. Items in lowercase must be replaced by the information you require. Alternatives are separated by the vertical stroke, |. Items enclosed in square brackets are optional. Underscored items are the default. If the command exceeds one line, the continuation marker - must be used on each line except the last.

The DATA and INDEX operands of the DEFINE CLUSTER command allow different attributes to be specified for the data component of the data set and the index component. This cannot usefully be done without more information than is available in this manual, and consequently the discussion is limited to the options of CLUSTER. However, separate specification of data and index components is important for VSAM operational control and efficient performance:

- Good VSAM data set naming conventions usually dictate separate names specified for the data and index components of a cluster, and this convention is especially useful with the Data Facility/Extended Function program product

installed (and extended-function catalogs in use), because in this case it is the component name(s) for which VTOC entries are created for VSAM. These VTOC entries are for the components — not the cluster.

- VSAM calculates control interval sizes for your data set, but it does so with the goal of optimizing disk space, not performance. It uses its calculated CI size for both data and index, when in fact the best values for the two types of CI size usually differ.

Thus most DEFINE CLUSTER commands should supply the DATA operand (for all types of clusters) and the INDEX operand (for a KSDS) with a user-supplied name and a value for CI-size for each component.

For more information, you should refer to the Access Method Services manual.

NAME(data-set-name)
The name may contain from 1 through 44 alphameric characters (A through Z, 0 through 9, @, #, and $), and two special characters (the hyphen and the 12-0 overpunch). Names containing more than 8 characters must be segmented by periods; 1 to 8 characters may be specified between periods. The first character of any name or name segment must be alphabetic.

VOLUMES(volser1 [volsern])
specifies the volume, or volumes on which your data set is to reside. For example, VOLUMES(HUR137 HUR138).

FILE(ddname)
specifies the name of the DD statement that identifies the device and volume to be used for space allocation. The DD statement you specify must have this syntax:

```
//ddname DD UNIT=(devtype[,unitcount]),
// VOL=SER=(volser1,volser2,...),...
```

REUSE|NOREUSE
specifies whether the cluster can be opened again and again as a temporary, or reusable, cluster. REUSE allows you to create an entry-sequenced, key-sequenced, or relative-record workfile.

When you create a reusable cluster, you cannot build an alternate index to support it. Also, you cannot create a reusable cluster with key ranges or with its own data space. Reusable data sets may be multivolumed and are restricted to 16 physical extents per volume.

INDEXED|NONINDEXED|NUMBERED
specifies the type of VSAM data set as follows:

INDEXED	Key-sequenced data set
NONINDEXED	Entry-sequenced data set
NUMBERED	Relative record data set

INDEXED is the default.

KEYS(length offset)
applies to key sequenced data sets only and specifies the position and length of the key. In VSAM, all keys are within the record.

length
is the length of the key in bytes.

offset
is the offset from the start of the record.

For example, KEYS(10 0) means that the first 10 characters (bytes) of the record are to be used as a key.

FREESPACE(ci% ca%)
specifies the amount of space that will be left empty in a key sequenced data set. Free space can be left to allow for expansion of the data set in a way that will not degrade the speed of sequential access.

ci%
is the percentage of each control interval that is to be left empty.

ca%
is the percentage of control intervals in each control area to be left empty.

Control intervals are collections of records. Control areas are collections of control intervals. The sizes are determined by VSAM to suit the devices used, or the control interval size can be specified in the DEFINE command (see the Access Method Services manual).

The default is FREESPACE(0 0).

TRACKS|CYLINDERS|RECORDS (primary-alloc secondary-alloc)
specifies the space that is to be reserved for your data set, in either tracks, cylinders, or records. The primary allocation is reserved when the DEFINE CLUSTER command is executed. The secondary allocation is reserved when the primary allocation has been filled. Up to 16 secondary allocations can be made.

RECORDSIZE(average maximum)
specifies the size of records. Average size and maximum size must be specified in bytes. For relative record data sets, fixed-length records are required; consequently average and maximum must be the same. For other types of data sets, records can be any length less than or equal to the maximum length.

password-options
specify the password or passwords for your data set. The options are as follows:

MASTERPW(password)
gives complete access to data set.

CONTROLPW(password)
is irrelevant to PL/I users.

UPDATEPW(password)
gives access to alter contents.

READPW(password)
gives read-only access.

Password is 1 to 8 EBCDIC characters.

If only a low level password, such as READPW, is specified, the read password is propagated upwards so that it also becomes the other passwords. If only a high-level password is specified, lower level passwords will not be required.

SHAREOPTIONS(n[m])
is described below, under "Sharing a Data Set between Jobs" on page 390.

other-options
Numerous other options can be specified that control the physical structure, data integrity, and protection of VSAM data sets. See the Access Method Services manual.

CATALOG(catname/password)
specifies the name and password, if any, of the catalog in which the data set is to be defined. If CATALOG is omitted, the master catalog is the default. If you have not yet defined your catalog, see the Access Method Services manual.

An example of the use of the DEFINE CLUSTER command is:

```
DEFINE CLUSTER (NAME(BOB) -
        VOLUMES(HUR136) -
        INDEXED -
        KEYS(10 20) -
        FREESPACE(20 10) -
        RECORDSIZE(50 80) -
        TRACKS(20 5)) -
      CATALOG(CRIPPEN/BORDEN)
```

This defines a key sequenced data set called BOB on the volume HUR136. The key is 10 bytes long and starts at offset 20 (the 21st character). 20% of each control interval, and 10% of the control intervals in each control area will be kept empty for new records. The primary space allocation is 20 tracks and secondary allocations will be in increments of 5 tracks. The catalog in which the data set is to be defined is called CRIPPEN and the password is BORDEN.

Complete examples of PL/I statements, JCL, and Access Method Services commands are given in Chapter 7, "Using VSAM Data Sets from PL/I" on page 222.

USING THE ACCESS METHOD SERVICES PROGRAM

How you use the Access Method Services program depends on whether you work in a batch or interactive system. In a batch environment, you execute the program as a separate job or job step, supplying the command in the SYSIN data set, and providing a SYSPRINT data set for printing any messages. For example:

```
//FRED  JOB  ....
//    EXEC  PGM=IDCAMS
//SYSPRINT DD  SYSOUT=A
//SYSIN    DD   *
     DEFINE CLUSTER (NAME(FRED) -
             VOLUMES(HUR137) -
             TRACKS(10 5) -
             RECORDSIZE(80 100) -
             NONINDEXED) -
          CATALOG(MASTCAT)
/*
```

SHARING VSAM DATA SETS

VSAM data sets can be shared within a job, between jobs (cross-region sharing), and between two or more operating systems (cross-system sharing). The extent to which they may be shared depends upon the SHAREOPTIONS specified in the DEFINE command when the data set is defined. For information on sharing data sets between systems, read the Access Method Services manual, bearing in mind that PL/I does not enable you to issue the RESERVE, RELEASE, ENQ, or DEQ macro instructions. A short description of sharing between jobs and sharing within a job follows. Again, full information is given in the Access Method Services manual.

SHARING A DATA SET BETWEEN JOBS

When issuing the DEFINE CLUSTER command, it is possible to use the SHAREOPTIONS parameter to specify the amount of sharing that will be allowed on the data set. The option is specified with a number, n, or two numbers, n and m, separated by a blank, with the syntax shown above, where:

n specifies cross-region sharing and has the following meanings:

n=1
specifies that any number of users can share the component or cluster being defined if only read operations are being performed. When a write operation is being performed, only one user at a time can use the component or cluster.

n=2
specifies that any number of users can use the component or cluster for read operations even if one user is using it for a write operation.

n=3
specifies that any number of users can share the component or cluster for both read and write operations. Data integrity is the users' responsibility, and VSAM provides no assistance in maintaining it.

n=4
specifies that any number of users can share the component or cluster for both read and write operations. Data integrity is the users' responsibility, but VSAM provides some assistance. This option requires your program to use the ENQ and DEQ macros to maintain data integrity while sharing the data set. PL/I does not issue ENQ or DEQ.

m specifies cross-system sharing, as described in the Access Method Services manual.

When a data set is opened, VSAM checks to see if it is being shared and if it is, whether the type of sharing is allowed in the SHAREOPTIONS. If it is not allowed, the file is not opened. To share a data set, each user must specify DISP=SHR in the data set's DD statement. The use of DISP=OLD in the DD statement causes the share options to be set to 1 3, to ensure that you have sole control of the data set except for the case of cross-system sharing.

SHARING WITHIN A JOB

Data sets can be shared within a job by having a number of DD statements specifying the same data set, or by opening the data set by a number of alternate index paths, or by both methods at once. Generally speaking, there are no restrictions on this type of use. However, it is possible for errors to occur when one file is holding a control interval and the same control interval is required by another file. Such errors can be avoided by not having two files associated with the same data set at one time.

DELETING A VSAM DATA SET

To delete a VSAM data set you need to know:

- The name of the data set
- Its master password, if any, or the master password of the catalog that contains it
- The name of the catalog in which it is placed if it is not in the master catalog
- How to code and use the DELETE subcommand

VSAM data sets are deleted by the DELETE command of Access Method Services:

Syntax

```
DELETE (data-set-name[/masterpw) -
       [CATALOG(catname[/masterpw]) -
       [other-options]
```

data-set-name
is the name of the data set that you want to delete. masterpw is the master password for the data set.

CATALOG(catname[/masterpw])
specifies the name of the catalog on which the data set is cataloged. If it is the master catalog, CATALOG can be omitted.

masterpw is the master password of the catalog and is required only if the data set is password protected and the data set password is not specified with the data set name.

other-options
specify other facilities of the DELETE command. These are described in the Access Method Services manual.

An example of deleting a data set in a batch programming environment is:

```
//DELET  JOB .....
//    EXEC  PGM=IDCAMS
//SYSPRINT DD   SYSOUT=A
//SYSIN    DD   *
   DELETE FRED CATALOG(MASTCAT)
/*
```

This deletes the data set FRED defined in the example of the DEFINE command shown earlier in this section. See "Using the Access Method Services Program" on page 389 for a fuller description of using Access Method Services, including descriptions of interactive environments.

ALTERNATE INDEX PATHS

VSAM allows alternate indexes to be defined on key sequenced and entry sequenced data sets. This enables key sequenced data sets to be accessed in a number of ways apart from use of the prime index, and allows entry sequenced data sets to be indexed and accessed 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 non-unique key. For a unique key alternate index, each record must have a different key. For a non-unique key alternate index, any number of records can have the same 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), and the alternate index using the department number would be a non-unique 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 93 on page 225.

A data set accessed through a unique key alternate index path can be treated, in most respects, like a KSDS accessed through its prime index. The records may be accessed by key or sequentially, records may be updated, and new records may be added. If the data set is a KSDS, records may be deleted and the length of updated records altered. Restrictions and allowed processing are shown in Figure 95 on page 228. When records are added or deleted, all indexes associated with the data set are by default altered to reflect the new situation.

In data sets accessed through a non-unique 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 may 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.

HOW TO BUILD AND USE ALTERNATE INDEX PATHS

If you are using alternate indexes, knowledge of how to use them is required at four stages of the programming process, as it is with normal data sets. These stages are:

1. When planning and coding the program
2. When creating the alternate indexes
3. When executing the program that accesses the data set through the alternate indexes
4. When deleting the alternate index, if you wish to delete it at a different time from the associated data set

Discussions of what to do at these stages follow, but are preceded by a short section on the terminology used with alternate indexes.

Terminology

An alternate index is, in practice, a VSAM data set that contains a series of pointers to the keys (or their equivalent) of a VSAM data set. When you use an alternate index to access a data set you should use a third entity known as an alternate index path or simply a path, that establishes the relationship between the index and the data set.

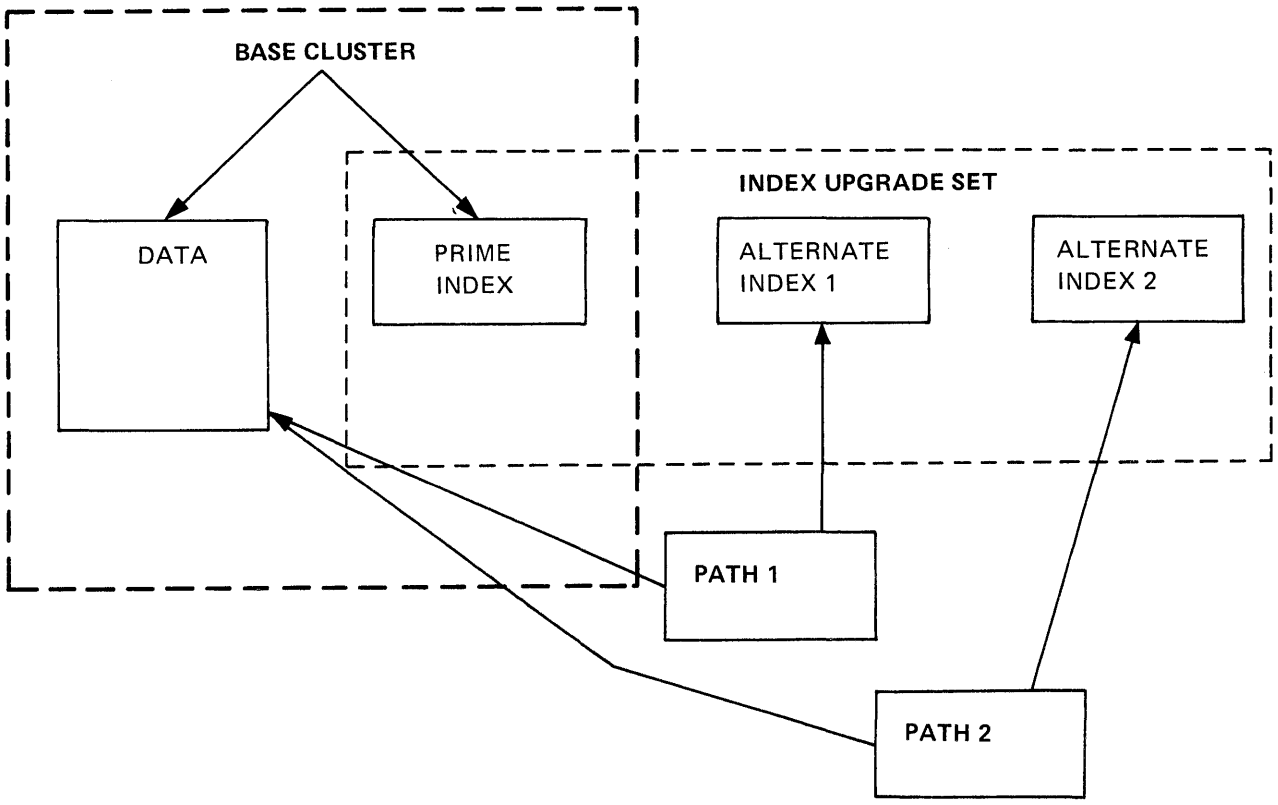
The data set to which the alternate index gives you access is known as the base data set, or more usually in the VSAM manuals as the base cluster.

The indexes of a base cluster are, by default, connected to it in such a way that alteration to the data will be reflected in the indexes. All indexes so connected are known as the index upgrade set of the base cluster. The relationship between the items is shown in Figure 158 on page 393.

PLANNING AND CODING WITH ALTERNATE INDEXES

When planning to use an alternate index you must know:

- The type of base data set with which the index will be associated
- Whether the keys will be unique or non-unique
- Whether the index is to be password protected
- Some of the performance aspects of using alternate indexes



Base Cluster	Accesses data by prime index (except for ESDS).
Prime index	Is the index used in creating the data set and used when access is made through the base cluster.
Alternate Indexes	Are other indexes to the same base data.
Paths	Establish a path through the base data other than that implied by the prime index in a KSDS and the sequence in an ESDS. Paths connect the alternate index with the base data.
Index upgrade set	That set of indexes (always including the prime index) that will be automatically updated when the data is changed. Note that indexes can exist outside this set.

Figure 158. Base Cluster, Alternate Indexes, and Paths

The type of the base cluster and the use of unique or non-unique keys determine the type of processing that you can carry out with the alternate index, and so determine the PL/I statements you may use. Figure 94 on page 228 and Figure 95 on page 228 show respectively the basic file attributes that you can use with an alternate index path and the types of processing that you can use.

Broadly, you use an alternate index path just like any other data set. In fact, a PL/I file could be used to access a data set directly in one execution and used to access a data set via an alternate index path in another.

Passwords

The alternate index may be password protected, as for a normal VSAM data set.

Performance

Performance with alternate indexes is not significantly worse than performance using the prime index. However, as the use of alternate indexes introduces an additional level of indirection into the access of a record, access cannot be as fast.

When updating a data set with more than one index, the resulting index upgrade also degrades performance.

When opening a data set with a number of indexes, the indexes are by default opened at the same time as the data set to allow for possible upgrade. This overhead can be avoided when the data set is being used for read-only processing by specifying the NOUPDATE attribute on the DEFINE PATH command. By specifying NOUPDATE, only the base cluster, which is the key-sequenced or entry-sequenced data set, is changed. The method of defining such a path is described in the MVS/Extended Architecture VSAM Administration Guide. When using the DEFINE PATH command, be careful not to alter the data set. One way to avoid changing the data set is to use the UPDATEPW attribute.

HOW TO BUILD AN ALTERNATE INDEX

To build and use an alternate index, you issue three Access Method Services commands:

```
DEFINE ALTERNATEINDEX
BLDINDEX
DEFINE PATH
```

DEFINE ALTERNATEINDEX defines and catalogs the data set that will hold the alternate index, and associates it with the base cluster. BLDINDEX reads the base cluster, extracts the keys, sorts them, and builds the alternate index by inserting pointers to the records. DEFINE PATH establishes a path that you will be able to associate with your PL/I file when you want to access the base data set through the alternate index. An alternate index cannot be built unless there are records in the data set.

To use these commands you will need to know:

- The name of the base data set
- The password for the base data set, if any
- The position and length of the alternate index key in the record
- The approximate size of the base cluster
- Whether the keys will be unique or non-unique
- If the keys will be non-unique, the approximate maximum number of records with the same key
- The catalog on which the alternate index is to be placed

When you have established these facts, you are in a position to code and execute the commands.

The commands must be issued in the order shown. A separate job step must be used for BLDINDEX and DEFINE PATH. An example showing the commands in one jobstep is given at the end of this section.

DEFINE ALTERNATEINDEX Command

The simplified syntax of the DEFINE ALTERNATEINDEX command follows:

Syntax

```
DEFINE ALTERNATEINDEX -
  (NAME(indexname) -
  VOLUMES(volser) -
  TRACKS|CYLINDERS|RECORDS -
   (primary-alloc  secondary-alloc) -
  KEYS(length offset) -
  UNIQUEKEY|NONUNIQUEKEY -
  UPGRADE|NOUPGRADE -
  RELATE(base-data-set/masterpw) -
  RECORDSIZE(average maximum) -
  [MASTERPW(password) -
  CONTROLPW(password) -
  UPDATEPW(password) -
  READPW(password) -
  other-options]) -
CATALOG(catname/password)
```

Note: Only those options that are different from those of the DEFINE CLUSTER command are explained below. If in doubt about the others, see "DEFINE CLUSTER Command" on page 386.

NAME(indexname)
specifies the name of the alternate index. It can be any name allowed for an OS data set. (See DEFINE CLUSTER command.)

KEYS(length offset)
specifies the position of the alternate index key in the record. They may be anywhere within the record. (For spanned records, all keys must be in the first section of the record.)

UNIQUEKEY|NONUNIQUEKEY
specifies whether the keys will be unique. If duplicate keys are found when building an alternate index that has been given the UNIQUEKEY attribute, an error occurs and the execution of BLDINDEX is halted.

UPGRADE|NOUPGRADE
specifies whether the alternate index is to be part of the index upgrade set for the data set. If it is, it is updated whenever the base data set is altered (using this index or any other index). If NOUPGRADE is specified, the index is not altered with the data set.

RELATE(base-data-set/masterpw)
specifies the base data set with which the alternate index will be associated.

RECORDSIZE(average maximum)
specifies the size of the record in the alternate index. If the path is non-unique, each index record will have to refer to many data records. Consequently, if the key is non-unique, the maximum should be a large figure. The default values are large; see the Access Method Services manual.

MASTERPW, CONTROLPW, UPDATEPW, READPW
specify the password options of the alternate index. See DEFINE CLUSTER command for details.

other-options
are described in the Access Method Services manual.

CATALOG(catname/password)
specifies the catalog in which the alternate index will be defined. It must be the same as the catalog of the base data set.

An example of the DEFINE ALTERNATEINDEX command is:

```
DEFINE ALTERNATEINDEX -
    (NAME(ALPHINDX) -
    VOLUMES(HUR137) -
    KEYS(10 0) -
    NONUNIQUEKEY -
    RELATE(PERSNOS) -
    RECORDSIZE(20 2000)) -
CATALOG(CRIPPEN/BORDEN)
```

This defines an alternate index called ALPHINDX on the data set called PERSNOS. The keys are non-unique and are in the first 10 bytes of the record. It is cataloged in the catalog called CRIPPEN with the master password BORDEN.

BLDINDEX Command

The BLDINDEX command extracts keys from the base data set, sorts them into order, and places the necessary information in the alternate index. DD statements, or their equivalent, are required for the base cluster, the alternate index, and two work files that may be needed if the necessary sorting cannot be carried out in main storage.

Syntax

```
BLDINDEX  INFILE(ddname1[/read-password]) -
 OUTFILE(ddname2[/update-password]) -
 CATALOG(catname[/update-password])
```

where ddname1 is the ddname of the base data set and ddname2 is the ddname of the alternate index. For example:

```
BLDINDEX INFILE(BASE) -
    OUTFILE(ALTIND) -
    CATALOG(CRIPPEN/BORDEN)
```

The DD statements use the following syntax:

Base Cluster:

```
//ddname1 DD DSNAME=base cluster,
//           DISP=OLD
```

For example:

```
//BASE   DD  DSNAME=PERSNOS,DISP=OLD
```

Alternate Index:

```
//ddname2 DD DSN=alternate index name,
//         DISP=OLD
```

For example:

```
//ALTIND  DD  DSNAME=ALPHIND,DISP=OLD
```

Sort Workfiles:

```
//IDCUT1 DD   DISP=OLD,AMP='AMORG',
//       VOL=SER=volno,UNIT=device type
//IDCUT2 DD   DISP=OLD,AMP='AMORG',
//       VOL=SER=volno,UNIT=device type
```

Note that UNIT must specify the generic device type of the volume or a unit address (SYSDA is not allowed). The volume must be a VSAM volume and space will be allocated for the workfiles by VSAM. AMP='AMORG' is required to indicate that space will be acquired by VSAM.

A combined example showing all the commands and job control statements to create an alternate index is given in Figure 159 on page 399.

DEFINE PATH Command

The DEFINE PATH command defines a name of the alternate index/base cluster combination, and enables it to be used from a PL/I program.

Syntax

```
DEFINE PATH (NAME(pathname) -
 PATHENTRY -
 (alternate-index-name[/masterpw]) -
 [MASTERPW(password) -
 CONTROLPW(password) -
 UPDATEPW(password) -
 READPW(password) -
 other-options]) -
[CATALOG(catname[/masterpw])]
```

other-options are described in the Access Method Services manual.

The master password of the catalog, which must be the same catalog as that used by the base cluster and the alternate index, is an alternative to the use of the master password of the base cluster.

An example of the DEFINE PATH command is:

```
DEFINE PATH -
    (NAME(ALPHPERS) -
     PATHENTRY(ALPHIND)) -
  CATALOG(MASTCAT)
```

EXECUTING THE ACCESS METHOD SERVICE COMMANDS TO CREATE AN ALTERNATE INDEX PATH

The example in Figure 159 on page 399 shows the use of Access Method Services in a batch system. If you use TSO, the Access Method Services commands are issued as TSO commands with ALLOCATE commands used instead of DD statements. If you use CMS, the Access Method Services Commands are written in a file with the file type AMSERV, and the name of the file specified in an AMSERV command.

In the example, the existence of a data set PERSNOS which contains data records is assumed. It is a data set keyed by personnel numbers. An alternate index called ALPHIND is being generated on the data set keyed by the first 25 characters of the records that contain the name. The path that specifies the base data set/alternate index pair is to be called PERSALPH. The catalog used by all items is NMCAT, and the volume HUR137.

The example is commented to simplify understanding. Access Method Services comments are delimited by /* and */. JCL comments are one line in length and start with //*. These are the allowed syntaxes for comments.

DELETING AN ALTERNATE INDEX

Alternate indexes and alternate index paths are deleted when the associated base data set is deleted. If you want to delete them separately without deleting the base data set, you specify them in the DELETE command. For example:

To delete an alternate index:

```
DELETE(ALTIND/SESAME)
```

where ALTIND is the name of the alternate index and SESAME is the master password.

To delete a path:

```
DELETE(ALTPATH/SESAME)
```

where ALTPATH is the name of the path and SESAME is the password.

```
//ALT  JOB  .....
//STEP1  EXEC  PGM=IDCAMS
//SYSIN  DD  *

     DEFINE ALTERNATEINDEX -
         (NAME(ALPHIND)        /*DSNAME of alternate index*/ -
          VOLUMES(HUR137)      /*volume on which it is placed*/ -
          TRACKS(10,1)         /*space used by alternate index*/ -
          NONUNIQUEKEY         /*keys will not be unique*/ -
          RECSIZE(20 1000)     /*average will be one personnel -
                                 number per name but some names will have -
                                 many numbers so large maximum required*/ -
          RELATE(PERSNOS))     /*name of associated data set*/ -
       CATALOG(NMCAT)   /*catalog name must be same as base data set's*/
//STEP2  EXEC PGM=IDCAMS
//* DD statements for BLDINDEX command follow
//* first the alternate index
//ALTIND  DD  DSNAME=ALPHIND,DISP=OLD
//* then the base data set
//BASEDS  DD   DSNAME=PERSNOS,DISP=OLD
//* the DD statements for BLDINDEX sort files follow
//IDCUT1  DD  DISP=OLD,AMP='AMORG',VOL=SER=HUR137,UNIT=3330
//IDCUT2  DD  DISP=OLD,AMP='AMORG',VOL=SER=HUR137,UNIT=3330
//SYSPRINT DD  SYSOUT=A
//SYSIN   DD  *

     BLDINDEX -
         /*this command loads the data into the alternate -
          index created in the previous command */ -
         INFILE(BASEDS)        /*dd name of base data set*/ -
         OUTFILE(ALTIND)       /*dd name of alternate index*/ -
       CATALOG(NMCAT)

     DEFINE PATH -
         /*this command enables you to use alternate index -
           base cluster pair from your program */ -
             (NAME(PERSALPH)  /*name of alternate index path to be used -
                   as DSNAME in DD statement PL/I program*/ -
              PATHENTRY(ALPHIND)) /*name of alternate index*/ -
            CATALOG(NMCAT)
/*
```

In this example there are five names involved:

1. The DSNAME of the base data set—PERSNOS. Used in the RELATE operand of the DEFINE ALTERNATEINDEX command, and as the DSNAME in the DD statement for the INFILE of the BLDINDEX command.

2. The dd name of the base data set—BASEDS. Used in the INFILE operand of the BLDINDEX command and as the dd name in the DD statement for the INFILE.

3. The DSNAME of the alternate index—ALPHIND. Given in the NAME operand of the DEFINE ALTERNATEINDEX command, and used as the DSNAME in the DD statement for the BLDINDEX OUTFILE, and in the PATHENTRY operand of the DEFINE PATH command.

4. The dd name of the alternate index—ALTIND. Used in the OUTFILE operand of the BLDINDEX command and as the dd name in the DD statement for the OUTFILE.

5. The name of the alternate index path—PERSALPH. Given in the NAME operand of DEFINE PATH and that will be used as the DSNAME when the base data set is accessed through the alternate index paths.

Figure 159. The Commands Required to Create an Alternate Index Path

APPENDIX B. REQUIREMENTS FOR PROBLEM DETERMINATION AND APAR SUBMISSION

GENERAL INFORMATION

To enable IBM programming service personnel to analyze a problem, we must be able to reproduce it at the IBM programming service location. It is therefore essential to supply the source program with the APAR to enable the problem to be reproduced and analyzed. Faster resolution of the problem will be possible if the source program is reduced to the smallest, least complex form that still contains the problem.

If the APAR is being submitted as a result of a previous APAR that was returned, supply the additional requested documentation and be sure to indicate the number of the previous APAR.

If the APAR is an original APAR, the materials that are required to be submitted with the APAR are listed below; they fall into two categories: machine-readable information and listings. Submission of all the required materials will normally eliminate any need to return the APAR for additional information, resulting in a faster resolution of the problem.

MACHINE-READABLE INFORMATION

The machine-readable information must be supplied on a nonlabelled tape. Use IEBGENER to copy sequential data sets to the tape and IEBCOPY to place partitioned data sets on the tape; the JCL used to create the tape must accompany the APAR. Alternatively, if only one small sequential data set is involved, the machine-readable information may be supplied as a deck of punched cards.

The machine-readable information should be carefully packed and clearly identified. As a minimum, ensure that the APAR number is present on the tape reel or card deck. This will allow the tape or deck to be identified should it become separated from the remainder of the material submitted with the APAR.

Three types of machine-readable information may be required, as detailed in the following sections.

ORIGINAL SOURCE

The term "original source" (as used here) is defined in one of the following three ways, depending on the type of problem:

1. If the compilation is performed with the NOINCLUDE and NOMACRO compiler options, the "original source" is the data set assigned to SYSIN for the compile step.

2. If the compilation is performed with either the INCLUDE or MACRO compiler options in effect and the problem is a preprocessor failure, the "original source" is the data set assigned to SYSIN for the compile step and the source statement library or libraries referenced in %INCLUDE statements in the program.

3. If the compilation is performed with either the INCLUDE or MACRO compiler options in effect and the problem is not a preprocessor failure, the "original source" is the SYSPUNCH data set produced by the compiler when the MDECK compiler option is specified.

The "original source" should have no %NOPRINT statements, unless they are relevant to the problem.

If the "original source," as defined above, is not supplied in the original submission of the APAR, the APAR will normally be returned.

LOAD LIBRARIES

If the failure occurs at execution-time and the source supplied calls one or more previously compiled modules, the load libraries containing these modules must be supplied in machine-readable form.

INPUT DATA SETS

If the failure occurs at execution-time, provide enough input data to allow the re-creation of the failure.

LISTINGS

All listings that are supplied must relate to a particular execution of the compiler, in the case of a suspected compiler failure, or to the relevant link-editing and execution steps, in the case of an execution-time failure. Listings derived from separate compilations or executions are of no value and may, in fact, be misleading to the programming support personnel.

Six types of listings may be required, as detailed in the following sections.

COMPILER LISTING

The listing which results from the compilation of the original source must accompany every APAR. Unless the opposite option is required to show the failure or unless the option masks the failure, the compilation must be performed with the following compiler options in effect:

```
ATTRIBUTES    LMESSAGE          NEST
DUMP          MAP               OPTIONS
FLAG (I)      MARGINI ('|')     SOURCE
LIST                            XREF
```

If the original source is the second type defined above, the INSOURCE compiler option must also be specified.

If the problem is an execution-time problem, the GOSTMT compiler option must also be specified.

(If any of the compiler options listed above have been deleted at system generation, they may be restored for temporary use by means of the CONTROL compiler option.)

JCL LISTING

Listings of job control statements used to run the program must be supplied. For batch jobs, any cataloged procedures must be shown in expanded form by specifying MSGLEVEL=(1,1) in the JOB statement.

CMS TERMINAL SESSION LOG

If the failure occurs while compiling or executing a program under CMS, full details of the virtual machine environment must be supplied. This can best be done as follows:

1. Immediately before invoking the compiler to reproduce the problem, issue the following commands:

```
QUERY SET
QUERY TERMINAL
QUERY VIRTUAL
QUERY SEARCH
QUERY DISK *
QUERY FILEDEF
QUERY LIBRARY
QUERY INPUT
QUERY OUTPUT
```

2. Invoke the compiler using the PLIOPT command, specifying the compiler options listed in the previous section, "Compiler Options" on page 11., and any other options required to produce the relevant output, preferably on a line printer, or, alternatively, at a typewriter terminal.

The entire terminal listing, from LOGON to LOGOFF, should be submitted. If a display terminal is used, spool console input/output using the

```
CP SPOOL CONSOLE START
```

command to provide the full details of all input entered and of all responses received.

LINKAGE EDITOR LISTING

If the problem is an execution-time failure, a linkage editor map produced when the copy of the program that fails was link-edited is essential for the analysis of the storage dump that must also be obtained.

EXECUTION-TIME DUMP

If the problem occurs during the execution of a PL/I program, a storage dump must be supplied. If at all possible, a formatted PL/I dump produced by the PL/I error-handling facilities should be provided by including the following statement in an ERROR on-unit that will be entered when the program fails:

```
CALL PLIDUMP ('TFHB');
```

If, for some reason, a formatted PL/I dump cannot be obtained, supply a storage dump obtained by using the system SYSUDUMP or SYSABEND facilities or by using a stand-alone dump program.

APPLIED FIXES

A list of any program temporary fixes (PTFs) and local fixes applied to either the compiler or to its libraries must be supplied with the APAR. If no such fixes have been applied, please so indicate specifically.

MATERIALS CHECKLIST

The following checklist is provided to summarize the materials that must accompany an APAR:

Material Required	When Required
*Machine-readable Information	
Original Source	Always
Load Libraries	Execution-time problems only
Input Data Sets	Execution-time problems only
Listings:	
Compiler Listing	Always
JCL Listing	Always, except TSO or CMS
CMS Terminal Session Log	CMS only
Linkage Editor Listing	Execution-time problems only
Execution-Time Dump	Execution-time problems only
Applied PTFs and Fixes	Always

Figure 160. Summary of Requirements for APAR Submission

Note:

* If the machine-readable material is supplied on a tape reel, a listing of the JCL used to build the tape must also be submitted.

APPENDIX C. SHARED LIBRARY CATALOGED PROCEDURES

The **shared library** is a PL/I facility that allows an installation to load PL/I resident library modules into the link pack area (LPA) so that they are available to all PL/I programs. This reduces space overheads.

The resident library subroutines to be included in the shared library can be chosen by the installation; they must include the initialization routine, the error-handling routine, the open file routine, and all modules addressed from the TCA that are not identical for multitasking and nonmultitasking programs. Further details of the shared library are given in OS PL/I Optimizing Compiler: Execution Logic, OS PL/I Optimizing Compiler: Installation Guide, and OS PL/I Optimizing Compiler: Installation Guide for MVS.

The routines in the shared-library are held in link-pack-area modules. Each of the link-pack modules contains a number of library routines, and is headed by an addressing control block known as a **transfer vector**.

You can use the shared library by using standard IBM-supplied cataloged procedures and overriding the link-edit and loader procedure steps.

EXECUTION WHEN USING THE SHARED LIBRARY

Use of the shared library is specified by the linkage editor statement INCLUDE SYSLIB(PLISHRE).

A load module created for use with one shared library will not execute with a different shared library. You will have to link-edit the object module again, including the dummy transfer vector module for the different shared library.

If the FETCH statement is used, both load modules must use (or neither can use) the shared library option.

Remember that the linkage editor or loader require a large amount of main storage for external symbol dictionary tables while processing the dummy transfer vector module. If you specify SIZE=200K in the PARM field of your EXEC statement for the linkage editor or loader (and use a region or partition of equivalent size), you will get sufficient main storage for processing with the largest possible shared library.

Your PL/I program may take slightly longer to execute when using a shared library, because all library calls have to pass through the transfer vectors. However, your main storage requirements for a region will be greatly reduced if you have carefully selected your shared library modules to suit the operating environment.

MULTITASKING CONSIDERATIONS

An installation can specify that it does not require either the multitasking or the nonmultitasking modules in the shared library. However, both multitasking and nonmultitasking versions of the program region module will still be created. The module for the unwanted environment will be a dummy. This prevents problems should an INCLUDE PLISHRE statement be included in a program that is intended to run in the environment with no shared library. If this process was not carried out, such a statement could result in the incorrect environment being initialized.

USING STANDARD IBM CATALOGED PROCEDURES

Standard IBM-supplied cataloged procedures that use the linkage editor or loader (see Chapter 9, "Cataloged Procedures" on page 273) can be used to specify the shared library. This is done by overriding the SYSLIN DD statement in the link-edit or load-and-go procedure steps to ensure that the shared library addressing module is included.

For example, the cataloged procedure PLIXCL requires the following statements to make use of the shared library.

```
//STEP1       EXEC    PLIXCL
//LKED.SYSIN  DD      *
  INCLUDE     SYSLIB(PLISHRE)

  ENTRY       PLISTART

 (add further input here)

/*
```

You can add other linkage-editor control statements by placing them as indicated. For example, to give the resulting load module the name MINE, add the statement:

```
NAME      MINE(R)
```

between the ENTRY and /* statements.

APPENDIX D. SAMPLE PROGRAM

This appendix, consisting of a PL/I sample program, illustrates all the components of the listings produced by the compiler and the linkage editor. You may also use this sample program to verify that PL/I has been installed correctly on your system.

The listings themselves are described in Chapter 2, "The Compiler" on page 3 and Chapter 3, "The Linkage Editor and the Loader" on page 65.

The function of the program is fully documented in both the preprocessor input and the source listing by means of PL/I comments. These comments consist of lines of text each preceded by /* and followed by */. Note that the /* must not appear in columns 1 and 2 of the input record because it will be taken as a job control end of file statement.

Most pages of the listings contain brief notes explaining the contents of the pages.

OPTIONS SPECIFIED (1)

```
OBJECT,ND;

*PROCESS AG,A,C,ESD,GS,IS,LIST,M,MAP,MAR(2,72,1),NEST,OF,STG,SYN,X;      00050000
```

OPTIONS USED (2)

```
AGGREGATE       NOCOUNT         ATTRIBUTES(FULL)
COMPILE         NODECK          CHARSET(60,EBCDIC)
ESD             NOFLOW          FLAG(I)
GOSTMT          NOGONUMBER      LINECOUNT(55)
INSOURCE        NOGRAPHIC       MARGINS(2,72,1)
LIST            NOIMPRECISE     SEQUENCE(73,80)
LMESSAGE        NOINCLUDE       SIZE(2087812)
MACRO           NOINTERRUPT     XREF(FULL)
MAP             NOMARGINI
NEST            NONDECK
OBJECT          NONUMBER
OFFSET          NOOPTIMIZE
OPTIONS         NOTERMINAL
SOURCE
STMT
STORAGE
SYNTAX
```

Start of the compiler listing.

(1) List of options specified in the PARM parameter of the EXEC statement.

(2) List of options used, whether obtained by default, or by being specified explicitly.

PREPROCESSOR INPUT

```
LINE
   1    /***** PL/I SAMPLE PROGRAM. *****/                                    00100000

   2    %/*****************************************************************/ 00150000
   3     /*                                                               */ 00200000
   4     /* USES COMPILE-TIME PREPROCESSOR TO MODIFY PL/I (F) SOURCE FOR   */ 00250000
   5     /* USE WITH THIS COMPILER.   THE PREPROCESSOR STATEMENTS FOLLOWING */ 00300000
   6     /* COULD BE PLACED ON A LIBRARY AND USED TO MODIFY SEVERAL SOURCE  */ 00350000
   7     /* PROGRAMS BY MEANS OF THE PREPROCESSOR %INCLUDE STATEMENT. THEY  */ 00400000
   8     /* PERFORM THE FOLLOWING FUNCTIONS:                               */ 00450000
   9     /*                                                               */ 00500000
  10     /*  1. CONVERT CALLS TO FOLLOWING PL/I (F) IHE... ROUTINES TO THE  */ 00550000
  11     /*     EQUIVALENT NEW PL/I... ROUTINES-                          */ 00600000
  12     /*           IHEDUMP/J/C/T  TO  PLIDUMP,                         */ 00650000
  13     /*           IHESRTA/B/C/D  TO PLISRTA/B/C/D,                    */ 00700000
  14     /*           IHECKPS/T  TO  PLICKPT,                             */ 00750000
  15     /*           IHERESN/T  TO  PLIREST/PLICANC,                     */ 00800000
  16     /*           IHESARC/IHETSAC  TO  PLIRETC.                       */ 00850000
  17     /*                                                               */ 00900000
  18     /*  2. CHANGE FIRST DECLARE/DCL STATEMENT FOUND TO INCLUDE         */ 00950000
  19     /*     BUILTIN ATTRIBUTE FOR FOLLOWING BUILT-IN FUNCTIONS(WHICH   */ 01000000
  20     /*     DO NOT TAKE ARGUMENTS, AND SO ARE NOT IMPLICITLY DECLARED  */ 01050000
  21     /*     BUILTIN FOR THIS COMPILER - AS THEY WOULD BE FOR PL/I (F))- */ 01100000
  22     /*           DATE, TIME, ONCODE, ONCHAR, ONSOURCE, ONLOC,        */ 01150000
  23     /*           ONFILE, ONKEY, EMPTY, NULL.                         */ 01200000
  24     /*     NOTE: THE ONCOUNT BIF IS OMITTED FROM THIS LIST, & IS USED */ 01250000
  25     /*           LATER TO SHOW THE EFFECT OF NOT DECLARING IT BUILTIN. */ 01300000
  26     /*           ANY REFERENCES TO IHE--- ROUTINES MUST BE REMOVED    */ 01350000
  27     /*           FROM DECLARE STATEMENTS BEFORE THE SOURCE PROGRAM IS */ 01400000
  28     /*           PREPROCESSED, OTHERWISE FAILURES MAY OCCUR WHEN THE  */ 01450000
  29     /*           CONVERTED PROGRAM IS LINK-EDITED.                   */ 01500000
  30     /*                                                               */ 01550000
  31     /*  3. CHANGE 'NULLO' TO 'NULL' - THERE IS NO NULLO BUILTIN       */ 01600000
  32     /*     FUNCTION FOR THIS COMPILER; NULL MUST BE USED BOTH WITH    */ 01650000
  33     /*     POINTER AND OFFSET VARIABLES.                             */ 01700000
  34     /*                                                               */ 01750000
  35     /*****************************************************************/;01800000
```

Source statements for the sample program, exactly 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 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
  36      %       DCL   (IHEDUMP, IHEDUMJ, IHEDUMC, IHEDUMT, DECLARE, DCL,       01850000
  37                     IHECKPT, IHECKPS) ENTRY;                               01900000

  38      %       DCL   (IHESRTA, IHESRTB, IHESRTC, IHESRTD, IHEREST,           01950000
  39                     IHERESN, IHESARC, IHETSAC, NULLO) CHAR;                02000000

  40      %       DCL   COUNT FIXED;                                            02050000


  41      %       COUNT = 0                    /* FIRST-TIME-IN SWITCH.      */;02100000

  42      %       DEACTIVATE DECLARE, DCL      /* ENSURE MODIFIED STATEMENTS */;02150000
  43      %       ACTIVATE DECLARE,            /* ARE NOT RESCANNED DURING   */ 02200000
  44                       DCL NORESCAN        /* PREPROCESSOR REPLACEMENT.  */;02250000
```

```
LINE
  45       % DECLARE: DCL:                          /* GENERATE BUILTIN DECLARES. */ 02300000
  46                PROC RETURNS(CHAR);                                               02350000
  47                COUNT = COUNT + 1              /* COUNT = 1 IF 1ST TIME IN.  */;02400000
  48                IF COUNT = 1                                                      02450000
  49                  THEN RETURN('DCL (DATE,TIME,ONCHAR,ONSOURCE,ONCODE,' ||       02500000
  50                              'ONLOC,ONFILE,ONKEY,EMPTY,NULL) BUILTIN, ' ||     02550000
  51                              'CKPT_RETC FIXED BIN(31),');                      02600000
  52                  ELSE RETURN('DCL');                                           02650000
  53       %       END;                                                             02700000

  54       % IHEDUMP: IHEDUMJ: IHEDUMC: IHEDUMT: /* REPLACED BY CALL TO        */ 02750000
  55                PROC(ID#) RETURNS(CHAR)       /* PLIDUMP ROUTINE, INCLUDING */;02800000
  56                DCL ID# CHAR                  /* ORIGINAL ID(IF PRESENT).   */;02850000
  57                IF ID# = '' THEN RETURN('PLIDUMP');                             02900000
  58                           ELSE RETURN('PLIDUMP(''TFCA'',''' || ID# || ''')');02950000
  59       %       END;                                                             03000000

  60       % IHECKPS: IHECKPT:                     /* CHANGE TO PLICKPT. PL/I(F) */ 03050000
  61                PROC(ARG1, ARG2, ARG3, ARG4)  /* DEFAULTS GENERATED WHERE   */ 03100000
  62                    RETURNS(CHAR)             /* NO ARGUMENTS ORIGINALLY.   */;03150000
  63                DCL (ARG1, ARG2, ARG3, ARG4) CHAR;                              03200000
  64                IF ARG1 = '' THEN ARG1 = '''SYSCHK''';                          03250000
  65                IF ARG2 = '' THEN ARG2 = ' ''''';                               03300000
  66                IF ARG3 = '' THEN ARG3 = ' ''PS''';                             03350000
  67                IF ARG4 = '' THEN ARG4 = ' CKPT_RETC';                          03400000
  68                RETURN('PLICKPT(' || ARG1 || ',' || ARG2 || ','                 03450000
  69                                  || ARG3 || ',' || ARG4 || ')');               03500000
  70       %       END;                                                             03550000

  71       %       IHESRTA = 'PLISRTA'           /* REPLACE                    */;03600000
  72       %       IHESRTB = 'PLISRTB'           /*  CALLS TO                  */;03650000
  73       %       IHESRTC = 'PLISRTC'           /*   IHE---                   */;03700000
  74       %       IHESRTD = 'PLISRTD'           /*    ROUTINES                */;03750000
  75       %       IHEREST = 'PLIREST'           /*     BY                     */;03800000
  76       %       IHERESN = 'PLICANC'           /*      CALLS TO              */;03850000
  77       %       IHESARC = 'PLIRETC'           /*       PLI---               */;03900000
  78       %       IHETSAC = 'PLIRETC'           /*        ROUTINES. .         */;03950000

  79       %/*     THERE IS NO NULLO BUILTIN FUNCTION FOR THIS COMPILER;            04000000
  80               NULL MUST BE USED INSTEAD.                                    */;04050000
```

```
LINE
  81       %/* END OF PREPROCESSOR STATEMENTS;  SOURCE STATEMENTS FOLLOW HERE: */;04100000

  82       %       NULLO = 'NULL';                                               04150000


  83       SAMPLE:                                                               04200000
  84               PROC OPTIONS(MAIN);                                           04250000


  85               DECLARE (PDATE, PTIME) CHAR(6);                               04300000

  86               DECLARE CVAR CHAR(255) VAR;                                   04350000

  87               DCL     1  BINVAR,                                            04400000
  88                         2  RETCODE FIXED BIN(31,0),                        04450000
  89                         2  FBVAR FIXED BIN;                                04500000


  90               PDATE = DATE;                                                 04550000
  91               PTIME = TIME;                                                 04600000
  92               PUT SKIP EDIT('SAMPLE PROGRAM: DATE = ', PDATE, ', TIME = ',  04650000
  93                            PTIME) (A(23), P'99/99/99', A(9), P'Z9.99.99');  04700000
  94               RETCODE = 0101;                                               04750000


  95               ON ERROR                                                      04800000
  96             BEGIN;                                                          04850000
  97               CALL IHEDUMP;                                                 04900000


  98       /* THESE STATEMENTS ILLUSTRATE PREPROCESSOR REPLACEMENT AND USE OF    04950000
  99          BUILTIN FUNCTIONS.   THEY WILL NEVER BE EXECUTED.                */05000000

 100               CALL IHEDUMJ(127);                                            05050000
 101               CALL IHEDUMC(RETCODE);                                        05100000
 102               CALL IHEDUMT;                                                 05150000

 103               FBVAR = ONCODE;                                               05200000
 104               CVAR = ONCHAR;                                                05250000
 105               CVAR = ONSOURCE;                                              05300000
 106               CVAR = ONLOC;                                                 05350000
 107               CVAR = ONFILE;                                                05400000
 108               CVAR = ONKEY;                                                 05450000
```

```
LINE
 109      /* THIS STATEMENT, WHICH WILL NEVER BE EXECUTED, USES 'ONCOUNT' WHICH  05500000
 110         IS NEITHER EXPLICITLY NOR IMPLICITLY DECLARED BUILTIN.  THE EFFECT  05550000
 111         IS SHOWN IN THE ATTRIBUTE LISTING AND DIAGNOSTIC MESSAGES.        */05600000

 112              FBVAR = ONCOUNT;                                               05650000
 113        END;                                                                 05700000


 114      /* THIS IS A DUMMY PROCEDURE TO ILLUSTRATE OTHER PREPROCESSOR          05750000
 115         REPLACEMENTS/NON-IMPLICITLY DECLARED BUILTIN FUNCTIONS.             05800000
 116         IT WILL NEVER BE EXECUTED.                                        */05850000

 117      DUMMY:                                                                 05900000
 118              PROC;                                                          05950000

 119              DCL   AVAR AREA BASED(PVAR),                                   06000000
 120                    OVAR OFFSET(AVAR),                                       06050000
 121                    A ENTRY RETURNS(CHAR(80)),                               06100000
 122                    SIZE FIXED BIN(31,0);                                    06150000

 123              AVAR = EMPTY;                                                  06200000
 124              PVAR = NULL;                                                   06250000
 125              OVAR = NULLO;                                                  06300000

 126              CALL IHESRTA('ARG1', 'ARG2', SIZE, RETCODE);       /*  S     */06350000
 127              CALL IHESRTB('ARG1', 'ARG2', SIZE, RETCODE, A);    /*   O    */06400000
 128              CALL IHESRTC('ARG1', 'ARG2', SIZE, RETCODE, B);    /*    R   */06450000
 129              CALL IHESRTD('ARG1', 'ARG2', SIZE, RETCODE, A, B); /*     T  */06500000
 130              CALL IHECKPS('ARG1', 'ARG2', 'PS', RETCODE); /* CHECKPOINT   */06550000
 131              CALL IHECKPT; /* CHECKPOINT   */                               06600000
 132              CALL IHEREST;                                 /* FORCE RESTRT */06650000
 133              CALL IHERESN;                                 /* CANCEL CKPT  */06700000
 134              CALL IHETSAC(RETCODE);        /* SET RETURN CODE(TASKING)    */06750000

 135      A:      PROC RETURNS(CHAR(80));   END;                /*  DUMMY EXIT  */06800000
 136                                                            /*  PROCEDURES  */06850000
 137      B:      PROC(RECORD);   DCL RECORD CHAR(80);   END;   /*  FOR  SORT.  */06900000

 138              END DUMMY;                                                     06950000


 139              CALL IHESARC(RETCODE);        /* SET RETURN CODE(NONTASKING) */07000000
 140              PUT SKIP LIST('END SAMPLE PROGRAM');                           07050000


 141              END SAMPLE;                                                    07100000
```

```
PREPROCESSOR DIAGNOSTIC MESSAGES
 (1)  (2)  (3)
ERROR ID L   LINE    MESSAGE DESCRIPTION

WARNING DIAGNOSTIC MESSAGES

IEL0184I W   97      TOO FEW ARGUMENTS TO FUNCTION 'IHEDUMP'.    NULL STRINGS PASSED AS MISSING ARGUMENTS.
IEL0217I W   97      ARGUMENT LIST FOR PROCEDURE 'IHEDUMP' IS MISSING.    PROCEDURE INVOKED WITHOUT ARGUMENTS.
IEL0184I W   102     TOO FEW ARGUMENTS TO FUNCTION 'IHEDUMT'.    NULL STRINGS PASSED AS MISSING ARGUMENTS.
IEL0217I W   102     ARGUMENT LIST FOR PROCEDURE 'IHEDUMT' IS MISSING.    PROCEDURE INVOKED WITHOUT ARGUMENTS.
IEL0184I W   131     TOO FEW ARGUMENTS TO FUNCTION 'IHECKPT'.    NULL STRINGS PASSED AS MISSING ARGUMENTS.
IEL0217I W   131     ARGUMENT LIST FOR PROCEDURE 'IHECKPT' IS MISSING.    PROCEDURE INVOKED WITHOUT ARGUMENTS.

END OF PREPROCESSOR DIAGNOSTIC MESSAGES
```

Diagnostic messages generated by the preprocessor. All messages generated by the optimizing compiler (including the preprocessor) are documented in the publication OS Optimizing Compiler: Messages.

(1) "ERROR ID" This identifies the message as originating from the optimizing compiler (IEL), and gives the message number.

(2) "L" This is the severity level of the message.

(3) "LINE" This gives the number of the line in which the error occurred.

SOURCE LISTING

```
STMT LEV NT                                                                    (1)
                                                                                R

            /***** PL/I SAMPLE PROGRAM. *****/                                 00100000

  1      0  SAMPLE:                                                            04200000
                 PROC OPTIONS(MAIN);                                           04250000

  2    1 0         DCL (DATE,TIME,ONCHAR,ONSOURCE,ONCODE,ONLOC,ONFILE,ONKEY,EMPTY,04300000  2
                NULL) BUILTIN, CKPT_RETC FIXED BIN(31), (PDATE, PTIME) CHAR(6);   04300000  1

  3    1 0         DCL CVAR CHAR(255) VAR;                                       04350000  2

  4    1 0         DCL     1  BINVAR,                                            04400000  1
                         2  RETCODE FIXED BIN(31,0),                            04450000
                         2  FBVAR FIXED BIN;                                    04500000

  5    1 0         PDATE = DATE;                                                 04550000
  6    1 0         PTIME = TIME;                                                 04600000
  7    1 0         PUT SKIP EDIT('SAMPLE PROGRAM: DATE = ', PDATE, ', TIME = ',  04650000
                              PTIME) (A(23), P'99/99/99', A(9), P'Z9.99.99');   04700000
  8    1 0         RETCODE = 0101;                                               04750000

  9    1 0         ON ERROR                                                      04800000
                BEGIN;                                                          04850000
 10    2 0         CALL PLIDUMP;                                                 04900000  1

            /* THESE STATEMENTS ILLUSTRATE PREPROCESSOR REPLACEMENT AND USE OF   04950000
               BUILTIN FUNCTIONS.   THEY WILL NEVER BE EXECUTED.                */05000000

 11    2 0         CALL PLIDUMP('TFCA','127');                                   05050000  1
 12    2 0         CALL PLIDUMP('TFCA','RETCODE');                               05100000  1
 13    2 0         CALL PLIDUMP;                                                 05150000  1

 14    2 0         FBVAR = ONCODE;                                               05200000
 15    2 0         CVAR = ONCHAR;                                                05250000
 16    2 0         CVAR = ONSOURCE;                                              05300000
 17    2 0         CVAR = ONLOC;                                                 05350000
 18    2 0         CVAR = ONFILE;                                                05400000
 19    2 0         CVAR = ONKEY;                                                 05450000
```

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.

(1) Maximum depth of replacement.

```
    STMT LEV NT                                                                    R

                 /* THIS STATEMENT, WHICH WILL NEVER BE EXECUTED, USES 'ONCOUNT' WHICH  05500000
                    IS NEITHER EXPLICITLY NOR IMPLICITLY DECLARED BUILTIN.  THE EFFECT  05550000
                    IS SHOWN IN THE ATTRIBUTE LISTING AND DIAGNOSTIC MESSAGES.        */05600000

      20   2  0          FBVAR = ONCOUNT;                                              05650000
      21   2  0       END;                                                             05700000


                 /* THIS IS A DUMMY PROCEDURE TO ILLUSTRATE OTHER PREPROCESSOR          05750000
                    REPLACEMENTS/NON-IMPLICITLY DECLARED BUILTIN FUNCTIONS.           05800000
                    IT WILL NEVER BE EXECUTED.                                      */05850000

      22   1  0  DUMMY:                                                               05900000
                      PROC;                                                          05950000

      23   2  0          DCL   AVAR AREA BASED(PVAR),                                06000000  1
                               OVAR OFFSET(AVAR),                                    06050000
                               A ENTRY RETURNS(CHAR(80)),                            06100000
                               SIZE FIXED BIN(31,0);                                 06150000

      24   2  0          AVAR = EMPTY;                                               06200000
      25   2  0          PVAR = NULL;                                                06250000
      26   2  0          OVAR = NULL;                                                06300000  1

      27   2  0          CALL PLISRTA('ARG1', 'ARG2', SIZE, RETCODE);       /*  S     */06350000  1
      28   2  0          CALL PLISRTB('ARG1', 'ARG2', SIZE, RETCODE, A);    /*   O    */06400000  1
      29   2  0          CALL PLISRTC('ARG1', 'ARG2', SIZE, RETCODE, B);    /*    R   */06450000  1
      30   2  0          CALL PLISRTD('ARG1', 'ARG2', SIZE, RETCODE, A, B); /*     T  */06500000  1
      31   2  0          CALL PLICKPT('ARG1', 'ARG2', 'PS', RETCODE); /* CHECKPOINT   */06550000  1
      32   2  0          CALL PLICKPT('SYSCHK', '', 'PS', CKPT_RETC); /* CHECKPOINT   */06600000  1
                                                                                     06600000
      33   2  0          CALL PLIREST;                                 /* FORCE RESTRT */06650000  1
      34   2  0          CALL PLICANC;                                 /* CANCEL CKPT  */06700000  1
      35   2  0          CALL PLIRETC(RETCODE);        /* SET RETURN CODE(TASKING)    */06750000  1

      36   2  0  A:       PROC RETURNS(CHAR(80));   END;                /*  DUMMY EXIT  */06800000
                                                                       /*  PROCEDURES  */06850000
      38   2  0  B:       PROC(RECORD);   DCL RECORD CHAR(80);   END;  /*  FOR  SORT.  */06900000  1

      41   2  0          END DUMMY;                                                  06950000


      42   1  0          CALL PLIRETC(RETCODE);        /* SET RETURN CODE(NONTASKING) */07000000  1
      43   1  0          PUT SKIP LIST('END SAMPLE PROGRAM');                        07050000


      44   1  0          END SAMPLE;                                                 07100000
```

ATTRIBUTE AND CROSS-REFERENCE TABLE (FULL)

(1) DCL NO.	(3) IDENTIFIER	(4) ATTRIBUTES AND REFERENCES
36	A	(5) ENTRY RETURNS(CHARACTER (80)) 28,30
23	AVAR	BASED (PVAR) ALIGNED AREA (1000) 24
38	B	ENTRY RETURNS(DECIMAL /* SINGLE */ FLOAT (6)) 29,30
4	BINVAR	AUTOMATIC /* STRUCTURE */
2	CKPT_RETC	AUTOMATIC ALIGNED BINARY FIXED (31,0) 32
3	CVAR	AUTOMATIC UNALIGNED CHARACTER (255) VARYING 15,16,17,18,19
2	DATE	BUILTIN 5
22	DUMMY	ENTRY RETURNS(DECIMAL /* SINGLE */ FLOAT (6))
2	EMPTY	BUILTIN 24
4	FBVAR	/* IN BINVAR */ AUTOMATIC ALIGNED BINARY FIXED (15,0) 14,20
2	NULL	BUILTIN 25,26
2	ONCHAR	BUILTIN 15
2	ONCODE	BUILTIN 14
******** (2)	ONCOUNT	AUTOMATIC ALIGNED DECIMAL /* SINGLE */ FLOAT (6) 20
2	ONFILE	BUILTIN 18

Attributes 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 listed in alphabetic order.

(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.

DCL NO.	IDENTIFIER	ATTRIBUTES AND REFERENCES
2	ONKEY	BUILTIN 19
2	ONLOC	BUILTIN 17
2	ONSOURCE	BUILTIN 16
23	OVAR	AUTOMATIC ALIGNED OFFSET (AVAR) 26
2	PDATE	AUTOMATIC UNALIGNED CHARACTER (6) 5,7
********	PLICANC	BUILTIN 34
********	PLICKPT	BUILTIN 31,32
********	PLIDUMP	BUILTIN 10,11,12,13
********	PLIREST	BUILTIN 33
********	PLIRETC	BUILTIN 42 35
********	PLISRTA	BUILTIN 27
********	PLISRTB	BUILTIN 28
********	PLISRTC	BUILTIN 29
********	PLISRTD	BUILTIN 30
2	PTIME	AUTOMATIC UNALIGNED CHARACTER (6) 6,7
********	PVAR	AUTOMATIC ALIGNED POINTER 24,25

DCL NO.	IDENTIFIER	ATTRIBUTES AND REFERENCES
39	RECORD	/* PARAMETER */ UNALIGNED CHARACTER (80)
4	RETCODE	/* IN BINVAR */ AUTOMATIC ALIGNED BINARY FIXED (31,0) 8,42 27,28,29,30,31,35
1	SAMPLE	EXTERNAL ENTRY RETURNS(DECIMAL /* SINGLE */ FLOAT (6))
23	SIZE	AUTOMATIC ALIGNED BINARY FIXED (31,0) 27,28,29,30
********	SYSPRINT	EXTERNAL FILE PRINT 7,43
2	TIME	BUILTIN 6

AGGREGATE LENGTH TABLE

(1) DCL NO.	(2) IDENTIFIER	LVL	DIMS	OFFSET	(3) ELEMENT LENGTH.	TOTAL LENGTH.
4	BINVAR	1			6	6
	RETCODE	2			4	
	FBVAR	2		4	2	
			SUM OF CONSTANT LENGTHS			6

(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.

STORAGE REQUIREMENTS

(1) BLOCK, SECTION OR STATEMENT	(2) TYPE	(3) LENGTH	(HEX)	(4) DSA SIZE	(HEX)
*SAMPLE1	PROGRAM CSECT	2256	8D0		
*SAMPLE2	STATIC CSECT	876	36C		
SAMPLE	PROCEDURE BLOCK	522	20A	576	240
9	ON UNIT	666	29A	272	110
DUMMY	PROCEDURE BLOCK	832	340	240	F0
A	PROCEDURE BLOCK	106	6A	192	C0
B	PROCEDURE BLOCK	124	7C	200	C8

Storage requirements. This table gives the main storage requirements for the program. These quantities do not include the main storage that will be required by the resident and transient library subroutines that will be included by the linkage editor or loaded dynamically during execution.

(1) Name of the block, section, or number of the statement in the program.

(2) Description of the block, section, or statement.

(3) Length in bytes of the storage areas in both decimal and hexadecimal notation.

(4) Length in bytes of the dynamic storage area (DSA) in both decimal and hexadecimal notation.

EXTERNAL SYMBOL DICTIONARY

(1) SYMBOL	(2) TYPE	(3) ID	(4) ADDR	(5) LENGTH
PLISTART	SD	0001	000000	000050
*SAMPLE1	SD	0002	000000	0008D0
*SAMPLE2	SD	0003	000000	00036C
PLITABS	WX	0004	000000	
PLIXOPT	WX	0005	000000	
IBMBPOPT	WX	0006	000000	
PLIXHD	WX	0007	000000	
IBMBEATA	WX	0008	000000	
PLIFLOW	WX	0009	000000	
PLICOUNT	WX	000A	000000	
IBMBPIRA	ER	000B	000000	
IBMBPIRB	ER	000C	000000	
IBMBPIRC	ER	000D	000000	
PLICALLA	LD		000006	
PLICALLB	LD		00000A	
PLIMAIN	SD	000E	000000	000008
IBMBKCPC	ER	000F	000000	
IBMBKCPA	ER	0010	000000	
IBMBKCPB	ER	0011	000000	
IBMBKCPA	ER	0012	000000	
IBMBKCPA	ER	0013	000000	
IBMBKSTD	ER	0014	000000	
IBMBKSTA	ER	0015	000000	
IBMBKSTC	ER	0016	000000	
IBMBKSTA	ER	0017	000000	
IBMBKSTB	ER	0018	000000	
IBMBKSTA	ER	0019	000000	
IBMBKSTA	ER	001A	000000	
IBMBKDMA	ER	001B	000000	
IBMBPRCA	ER	001C	000000	
IELCGOG	SD	001D	000000	0000B6
IELCGOH	SD	001E	000000	0000A4
IBMBSEDA	ER	001F	000000	
IBMBSIOA	ER	0020	000000	
IBMBCCCA	ER	0021	000000	
IBMBCCSA	ER	0022	000000	
IBMBCEDB	ER	0023	000000	
IBMBCHFD	ER	0024	000000	
IBMBCODE	ER	0025	000000	
IBMBCTHD	ER	0026	000000	
IBMBCUID	ER	0027	000000	
IBMBEOCA	ER	0028	000000	
IBMBEOLA	ER	0029	000000	
IBMBJDTA	ER	002A	000000	
IBMBJTTA	ER	002B	000000	
IBMBOCLA	ER	002C	000000	
IBMBOCLC	WX	002D	000000	

External symbol dictionary.

(1) "SYMBOL" A list of all the external symbols that make up the object module.

(2) "TYPE" 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 all these terms are given in Chapter 4.

(3) "ID" All entries, except LD-type entries, are identified by a hexadecimal number.

(4) "ADDR" Address (in hexadecimal) of LD-type entries only.

(5) "LENGTH" Length in bytes (in hexadecimal) of SD, CM, and PR type entries only.

IBMBSAOA	ER	002E	000000	
IBMBSEDB	WX	002F	000000	
IBMBSEOA	ER	0030	000000	
IBMBSIOE	WX	0031	000000	
IBMBSIOT	WX	0032	000000	
IBMBSLOA	ER	0033	000000	
IBMBSPLA	ER	0034	000000	
IBMBSPOA	ER	0035	000000	
IBMBCKDD	ER	0036	000000	
IBMBSXCA	WX	0037	000000	
IBMBSXCB	WX	0038	000000	
IBMBSIST	WX	0039	000000	
SAMPLE	LD		000008	
SYSPINT	SD	003A	000000	000020

STATIC INTERNAL STORAGE MAP

(1)	(2)	(3)
000000	A0000368	PROGRAM ADCON
000004	00000008	PROGRAM ADCON
000008	00000078	PROGRAM ADCON
00000C	00000096	PROGRAM ADCON
000010	0000020C	PROGRAM ADCON
000014	0000026A	PROGRAM ADCON
000018	000004B0	PROGRAM ADCON
00001C	0000050E	PROGRAM ADCON
000020	000007EC	PROGRAM ADCON
000024	00000842	PROGRAM ADCON
000028	00000858	PROGRAM ADCON
00002C	000008C0	PROGRAM ADCON
000030	000008C0	PROGRAM ADCON
000034	000008C0	PROGRAM ADCON
000038	000008C0	PROGRAM ADCON
00003C	000008C0	PROGRAM ADCON
000040	000008C0	PROGRAM ADCON
000044	000008C0	PROGRAM ADCON
000048	00000000	A..IELCGOG
00004C	00000000	A..IELCGOH
000050	00000000	A..IBMBCCCA
000054	00000000	A..IBMBCCSA
000058	00000000	A..IBMBCEDB
00005C	00000000	A..IBMBCHFD
000060	00000000	A..IBMBCODE
000064	00000000	A..IBMBCTHD
000068	00000000	A..IBMBCUID
00006C	00000000	A..IBMBEOCA
000070	00000000	A..IBMBEOLA
000074	00000000	A..IBMBJDTA
000078	00000000	A..IBMBJTTA
00007C	00000000	A..IBMBOCLA
000080	00000000	A..IBMBOCLC
000084	00000000	A..IBMBSAOA
000088	00000000	A..IBMBSEDB
00008C	00000000	A..IBMBSEOA
000090	00000000	A..IBMBSIOE
000094	00000000	A..IBMBSIOT
000098	00000000	A..IBMBSLOA
00009C	00000000	A..IBMBSPLA
0000A0	00000000	A..IBMBSPOA
0000A4	00000000	A..IBMBCKDD
0000A8	2000	DED
0000AA	58000017	FED
0000AE	5400000814040680	FED
	0808000000006800	
	00680000	
0000C2	58000009	FED
0000C6	5400000814040680	FED
	0808000008007400	
	00740000	
0000DA	0001	CONSTANT
0000DC		
0000E0	0000021800170000	LOCATOR
0000E8	0000000000060000	LOCATOR..PDATE
0000F0	0000022F00090000	LOCATOR
0000F8	0000023800120000	LOCATOR
000100	0000000000040000	LOCATOR
000108	0000000000030000	LOCATOR
000110	0000000000070000	LOCATOR
000118	0000000000020000	LOCATOR
000120	0000000000000000	LOCATOR
000128	91E091E0	CONSTANT
00012C	00000001	CONSTANT
000130	00000065	CONSTANT
000134	46000000	CONSTANT
000138	00000010	CONSTANT
00013C	FF000000	CONSTANT
000140	00000000	A..DCLCB
000144	80000000	A..PDATE
000148	80000000	A..TEMP
00014C	00000000	A..DCLCB
000150	00000000	A..TEMP
000154	8000012C	A..CONSTANT
000158	80000000	A..RETCODE
00015C	00000000	A..ENTRY PLIRETC
000160	00000000	A..DCLCB
000164	00000000	A..TEMP
000168	8000012C	A..CONSTANT
00016C	00000000	A..ENTRY PLIDUMP
000170	00000000	A..TEMP
000174	80000000	A..TEMP
000178	00000000	A..TEMP
00017C	80000000	A..TEMP
000180	80000000	A..FBVAR
000184	80000000	A..TEMP
000188	00000000	A..TEMP
00018C	00000000	A..TEMP
000190	00000000	A..SIZE
000194	80000000	A..RETCODE
000198	00000000	A..ENTRY PLISRTA
00019C	00000000	A..TEMP
0001A0	00000000	A..TEMP
0001A4	00000000	A..SIZE
0001A8	00000000	A..RETCODE
0001AC	80000000	A..TEMP
0001B0	00000000	A..ENTRY PLISRTB
0001B4	00000000	A..TEMP
0001B8	00000000	A..TEMP
0001BC	00000000	A..SIZE
0001C0	00000000	A..RETCODE
0001C4	80000000	A..TEMP

Static Internal Storage Map. This is a storage 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.

```
0001C8  00000000            A..ENTRY PLISRTC
0001CC  00000000            A..TEMP
0001D0  00000000            A..TEMP
0001D4  00000000            A..SIZE
0001D8  00000000            A..RETCODE
0001DC  00000000            A..TEMP
0001E0  80000000            A..TEMP
0001E4  00000000            A..ENTRY PLISRTD
0001E8  00000000            A..TEMP
0001EC  00000000            A..TEMP
0001F0  00000000            A..TEMP
0001F4  80000000            A..RETCODE
0001F8  00000000            A..ENTRY PLICKPT
0001FC  00000000            A..TEMP
000200  00000000            A..TEMP
000204  00000000            A..TEMP
000208  80000000            A..CKPT_RETC
00020C  00000000            A..ENTRY PLIREST
000210  00000000            A..ENTRY PLICANC
000214  80000000            A..RETCODE
000218  E2C1D4D7D3C540D7    CONSTANT
        D9D6C7D9C1D47A40
        C4C1E3C5407E40
00022F  6B40E3C9D4C5407E    CONSTANT
        40
000238  C5D5C440E2C1D4D7    CONSTANT
        D3C540D7D9D6C7D9
        C1D4
00024A  E3C6C3C1            CONSTANT
00024E  F1F2F7              CONSTANT
000251  D9C5E3C3D6C4C5      CONSTANT
000258  C1D9C7F1            CONSTANT
00025C  C1D9C7F2            CONSTANT
000260  D7E2                CONSTANT
000262  E2E8E2C3C8D2        CONSTANT
000268  0C1600000000020C    STATIC ONCB
```

STATIC EXTERNAL CSECTS

```
000000  FFFFFFFC41201000    DCLCB
        02D70F0000000000
        000000140008E2E8
        E2D7D9C9D5E30000
```

VARIABLE STORAGE MAP

IDENTIFIER	LEVEL	OFFSET	(HEX)	CLASS	BLOCK
BINVAR	1	208	D0	AUTO	SAMPLE
RETCODE	1	208	D0	AUTO	SAMPLE
FBVAR	1	212	D4	AUTO	SAMPLE
CVAR	1	240	F0	AUTO	SAMPLE
CKPT_RETC	1	216	D8	AUTO	SAMPLE
PDATE	1	228	E4	AUTO	SAMPLE
PTIME	1	234	EA	AUTO	SAMPLE
OVAR	2	184	B8	AUTO	DUMMY
SIZE	2	188	BC	AUTO	DUMMY
PVAR	1	220	DC	AUTO	SAMPLE
ONCOUNT	1	224	E0	AUTO	SAMPLE

TABLES OF OFFSETS AND STATEMENT NUMBERS

WITHIN PROCEDURE SAMPLE

OFFSET (HEX)	0	8E	A4	C0	190	198	19C	1B4	1F2
STATEMENT NO.	1	5	6	7	8	9	42	43	44

WITHIN ON UNIT

OFFSET (HEX)	0	5E	68	B0	F8	102	118	142	18C	1D8	222	26C	28A
STATEMENT NO.	9	10	11	12	13	14	15	16	17	18	19	20	21

WITHIN PROCEDURE DUMMY

OFFSET (HEX)	0	5E	6E	76	7A	D2	13E	1AA	22A	296	2FC	306	310	328
STATEMENT NO.	22	24	25	26	27	28	29	30	31	32	33	34	35	41

WITHIN PROCEDURE A

OFFSET (HEX)	0	56
STATEMENT NO.	36	37

WITHIN PROCEDURE B

OFFSET (HEX)	0	68
STATEMENT NO.	38	40

This page intentionally left blank

OBJECT LISTING

(1) (2)

```
* COMPILER GENERATED SUBROUTINE IELCGOG
000000  50 E0 1 00C               ST    14,12(0,1)
000004  58 F0 1 014               L     15,20(0,1)
000008  91 10 1 011               TM    17(1),X'10'
00000C  47 10 7 014               BO    *+8
000010  96 04 C 002               OI    2(12),X'04'
000014  D2 03 1 008 F 04C         MVC   8(4,1),76(15)
00001A  D2 00 1 01C F 026         MVC   28(1,1),38(15)
000020  48 F0 F 050               LH    15,80(0,15)
000024  4B F0 E 002               SH    15,2(0,14)
000028  91 C0 E 001               TM    1(14),X'C0'
00002C  47 E0 7 040               BNO   *+20
000030  4B F0 E 002               SH    15,2(0,14)
000034  91 40 1 01C               TM    28(1),X'40'
000038  47 80 7 040               BZ    *+8
00003C  06 F0                     BCTR  15,0
00003E  06 F0                     BCTR  15,0
000040  12 FF                     LTR   15,15
000042  07 B6                     BCR   11,6
000044  96 40 1 010               OI    16(1),X'40'
000048  53 F0 D 04C               L     15,76(0,13)
00004C  50 F0 1 008               ST    15,8(0,1)
000050  BE F8 1 022               STCM  15,8,34(1)
000054  4A F0 E 002               AH    15,2(0,14)
000058  91 C0 E 001               TM    1(14),X'C0'
00005C  47 E0 7 070               BNO   *+20
000060  4A F0 E 002               AH    15,2(0,14)
000064  91 40 1 01C               TM    28(1),X'40'
000068  47 80 7 070               BZ    *+8
00006C  5A F0 7 0B0               A     15,176(0,7)
000070  4A F0 7 0B4               AH    15,180(0,7)
000074  54 F0 7 0AC               N     15,172(0,7)
000078  55 F0 C 00C               CL    15,12(0,12)
00007C  47 20 7 086               BH    *+10
000080  50 F0 D 04C               ST    15,76(0,13)
000084  07 F6                     BR    6
000086  50 00 1 01C               ST    0,28(0,1)
00008A  18 71                     LR    7,1
00008C  18 0F                     LR    0,15
00008E  58 10 D 04C               L     1,76(0,13)
000092  58 F0 C 048               L     15,72(0,12)
000096  05 EF                     BALR  14,15
000098  50 00 D 04C               ST    0,76(0,13)
00009C  50 10 7 008               ST    1,8(0,7)
0000A0  BE 08 1 022               STCM  0,8,34(1)
0000A4  18 17                     LR    1,7
0000A6  58 00 1 01C               L     0,28(0,1)
0000AA  07 F6                     BR    6
0000AC  FFFFFFF8                  DC    X'FFFFFFF8'
0000B0  00000002                  DC    X'00000002'
0000B4                            DC    AL2(7)

* END OF COMPILER GENERATED SUBROUTINE

* COMPILER GENERATED SUBROUTINE IELCGOH
000000  94 FB C 002               NI    2(12),X'FB'
000004  91 40 1 010               TM    16(1),X'40'
000008  47 10 7 084               BO    *+124
00000C  58 F0 1 014               L     15,20(0,1)
000010  50 50 1 01C               ST    5,28(0,1)
000014  58 E0 1 00C               L     14,12(0,1)
000018  48 50 F 050               LH    5,80(0,15)
00001C  4B 50 E 002               SH    5,2(0,14)
000020  91 C0 E 001               TM    1(14),X'C0'
000024  47 E0 7 038               BNO   *+20
000028  4B 50 E 002               SH    5,2(0,14)
00002C  91 40 F 026               TM    38(15),X'40'
000030  47 80 7 038               BZ    *+8
000034  06 50                     BCTR  5,0
000036  06 50                     BCTR  5,0
000038  40 50 F 050               STH   5,80(0,15)
00003C  58 50 F 04C               L     5,76(0,15)
000040  4A 50 E 002               AH    5,2(0,14)
000044  91 C0 E 001               TM    1(14),X'C0'
000048  47 E0 7 05C               BNO   *+20
00004C  4A 50 E 002               AH    5,2(0,14)
000050  91 40 F 026               TM    38(15),X'40'
000054  47 80 7 05C               BZ    *+8
000058  41 55 0 002               LA    5,2(5,0)
00005C  50 50 F 04C               ST    5,76(0,15)
000060  48 E0 1 020               LH    14,32(0,1)
000064  41 E0 E 001               LA    14,1(0,14)
000068  40 E0 1 020               STH   14,32(0,1)
00006C  40 E0 F 052               STH   14,82(0,15)
000070  58 50 1 01C               L     5,28(0,1)
000074  91 10 1 010               TM    16(1),X'10'
000078  07 86                     BCR   8,6
00007A  58 F0 7 09C               L     15,156(0,7)
00007E  18 E6                     LR    14,6
000080  07 FF                     BR    15
000082  07 F6                     BR    6
000084  58 F0 7 0A0               L     15,160(0,7)
000088  05 EF                     BALR  14,15
00008A  58 E0 1 008               L     14,8(0,1)
00008E  BF E8 1 022               ICM   14,8,34(1)
000092  50 E0 D 04C               ST    14,76(0,13)
000096  94 BF 1 010               NI    16(1),X'BF'
00009A  07 F6                     BR    6
00009C                            DC    AL4(0)
0000A0                            DC    AL4(0)
```

See note on next page.

Object listing. This is a listing of the machine instructions generated by the optimizing compiler from the PL/I source program.

(1) Machine instructions (in hexadecimal).

(2) Assembler-language form of the machine instructions.

```
* END OF COMPILER GENERATED SUBROUTINE

* STATEMENT NUMBER  1
000000                          DC    C' SAMPLE'
000007                          DC    AL1(6)

* PROCEDURE                            SAMPLE

* REAL ENTRY
000008  90 EC D 00C             STM   14,12,12(13)
00000C  47 F0 F 014             B     *+16
000010  00000000               DC    A(STMT. NO. TABLE)
000014  00000240               DC    F'576'
000018  00000000               DC    A(STATIC CSECT)
00001C  58 30 F 010             L     3,16(0,15)
000020  58 10 D 04C             L     1,76(0,13)
000024  58 00 F 00C             L     0,12(0,15)
000028  1E 01                  ALR   0,1
00002A  55 00 C 00C             CL    0,12(0,12)
00002E  47 D0 F 030             BNH   *+10
000032  58 F0 C 074             L     15,116(0,12)
000036  05 EF                   BALR  14,15
000038  58 E0 D 048             L     14,72(0,13)
00003C  18 F0                   LR    15,0
00003E  90 E0 1 048             STM   14,0,72(1)
000042  50 D0 1 004             ST    13,4(0,1)
000046  41 D1 0 000             LA    13,0(1,0)
00004A  50 50 D 058             ST    5,88(0,13)
00004E  41 60 D 0B8             LA    6,184(0,13)
000052  50 60 D 070             ST    6,112(0,13)
000056  D7 00 D 0B8 D 0B8       XC    184(1,13),184(13)
00005C  92 01 D 0B9             MVI   185(13),X'01'
000060  92 C0 D 000             MVI   0(13),X'C0'
000064  92 24 D 001             MVI   1(13),X'24'
000068  41 80 3 268             LA    8,616(0,3)
00006C  50 80 D 05C             ST    8,92(0,13)
000070  D2 03 D 054 3 128       MVC   84(4,13),296(3)
000076  05 20                   BALR  2,0

* PROLOGUE BASE
000078  D2 07 D 0C0 3 0E8       MVC   LOCATOR..PDATE(8),
                                      232(3)
00007E  41 90 D 0E4             LA    9,PDATE
000082  50 90 D 0C0             ST    9,LOCATOR..PDATE
000086  D2 07 D 0C8 3 0E8       MVC   LOCATOR..PTIME(8),
                                      232(3)
00008C  41 A0 D 0EA             LA    10,PTIME
000090  50 A0 D 0C8             ST    10,LOCATOR..PTIME
000094  05 20                   BALR  2,0

* PROCEDURE BASE

* STATEMENT NUMBER  5
000096  41 70 D 0E4                   LA    7,PDATE
00009A  50 70 3 144                   ST    7,324(0,3)
00009E  96 80 3 144                   OI    324(3),X'80'
0000A2  41 10 3 144                   LA    1,324(0,3)
0000A6  58 F0 3 074                   L     15,A..IBMBJDTA
0000AA  05 EF                         BALR  14,15

* STATEMENT NUMBER  6
0000AC  41 F0 D 230                   LA    15,560(0,13)
0000B0  50 F0 3 148                   ST    15,328(0,3)
0000B4  96 80 3 148                   OI    328(3),X'80'
0000B8  41 10 3 148                   LA    1,328(0,3)
0000BC  58 F0 3 078                   L     15,A..IBMBJTTA
0000C0  05 EF                         BALR  14,15
0000C2  D2 05 D 0EA D 230             MVC   PTIME(6),560(13)

* STATEMENT NUMBER  7
0000C8  41 70 D 208                   LA    7,520(0,13)
0000CC  50 70 3 150                   ST    7,336(0,3)
0000D0  41 10 D 208                   LA    1,520(0,13)
0000D4  50 10 D 200                   ST    1,512(0,13)
0000D8  92 20 D 219                   MVI   537(13),X'20'
0000DC  41 10 3 14C                   LA    1,332(0,3)
0000E0  58 F0 3 090                   L     15,A..IBMBSIOE
0000E4  05 EF                         BALR  14,15
0000E6  41 A0 2 094                   LA    10,CL.7
0000EA  41 E0 3 0E0                   LA    14,224(0,3)
0000EE  41 F0 3 0A8                   LA    15,168(0,3)
0000F2  58 10 D 200                   L     1,512(0,13)
0000F6  90 EF 1 000                   STM   14,15,0(1)
0000FA  05 AA                         BALR  10,10
0000FC  41 E0 D 0C0                   LA    14,192(0,13)
000100  41 F0 3 0A8                   LA    15,DED..PDATE
000104  90 EF 1 000                   STM   14,15,0(1)
000108  05 AA                         BALR  10,10
00010A  41 E0 3 0F0                   LA    14,240(0,3)
00010E  41 F0 3 0A8                   LA    15,168(0,3)
000112  90 EF 1 000                   STM   14,15,0(1)
000116  05 AA                         BALR  10,10
000118  41 E0 D 0C8                   LA    14,200(0,13)
00011C  41 F0 3 0A8                   LA    15,DED..PTIME
000120  90 EF 1 000                   STM   14,15,0(1)
000124  05 AA                         BALR  10,10
000126  47 F0 2 0F8                   B     CL.8
00012A                          CL.7  EQU   *
00012A  41 E0 3 0AA                   LA    14,170(0,3)
00012E  58 10 D 200                   L     1,512(0,13)
```

```
000132  58 70 3 048          L     7,A..IELCGOG
000136  05 67                BALR  6,7
000138  58 F0 3 084          L     15,A..IBMBSAOA
00013C  05 EF                BALR  14,15
00013E  58 70 3 04C          L     7,A..IELCGOH
000142  05 67                BALR  6,7
000144  05 AA                BALR  10,10
000146  41 E0 3 0AE          LA    14,174(0,3)
00014A  58 10 D 200          L     1,512(0,13)
00014E  50 E0 1 00C          ST    14,12(0,1)
000152  58 F0 3 088          L     15,A..IBMBSEDB
000156  05 EF                BALR  14,15
000158  05 AA                BALR  10,10
00015A  41 E0 3 0C2          LA    14,194(0,3)
00015E  58 10 D 200          L     1,512(0,13)
000162  58 70 3 048          L     7,A..IELCGOG
000166  05 67                BALR  6,7
000168  58 F0 3 084          L     15,A..IBMBSAOA
00016C  05 EF                BALR  14,15
00016E  58 70 3 04C          L     7,A..IELCGOH
000172  05 67                BALR  6,7
000174  05 AA                BALR  10,10
000176  41 E0 3 0C6          LA    14,198(0,3)
00017A  58 10 D 200          L     1,512(0,13)
00017E  50 E0 1 00C          ST    14,12(0,1)
000182  58 F0 3 088          L     15,A..IBMBSEDB
000186  05 EF                BALR  14,15
000188  05 AA                BALR  10,10
00018A  47 F0 2 094          B     CL.7
00018E                CL.8   EQU   *
00018E  58 10 D 200          L     1,512(0,13)
000192  58 F0 3 094          L     15,A..IBMBSIOT
000196  05 EF                BALR  14,15

* STATEMENT NUMBER  8
000198  58 F0 3 130          L     15,304(0,3)
00019C  50 F0 D 0D0          ST    15,BINVAR.RETCODE

* STATEMENT NUMBER  9
0001A0  92 0C D 0B8          MVI   184(13),X'0C'

* STATEMENT NUMBER  42
0001A4  41 70 D 0D0          LA    7,BINVAR.RETCODE
0001A8  50 70 3 158          ST    7,344(0,3)
0001AC  96 80 3 158          OI    344(3),X'80'
0001B0  1B 55                SR    5,5
0001B2  41 10 3 158          LA    1,344(0,3)
0001B6  58 F0 3 15C          L     15,348(0,3)
0001BA  05 EF                BALR  14,15

* STATEMENT NUMBER  43
0001BC  41 F0 D 208          LA    15,520(0,13)
0001C0  50 F0 3 164          ST    15,356(0,3)
0001C4  41 10 D 208          LA    1,520(0,13)
0001C8  50 10 D 200          ST    1,512(0,13)
0001CC  92 40 D 219          MVI   537(13),X'40'
0001D0  41 10 3 160          LA    1,352(0,3)
0001D4  58 F0 3 090          L     15,A..IBMBSIOE
0001D8  05 EF                BALR  14,15
0001DA  41 E0 3 0F8          LA    14,248(0,3)
0001DE  41 F0 3 0A8          LA    15,168(0,3)
0001E2  58 10 D 200          L     1,512(0,13)
0001E6  90 EF 1 000          STM   14,15,0(1)
0001EA  58 F0 3 098          L     15,A..IBMBSLOA
0001EE  05 EF                BALR  14,15
0001F0  58 10 D 200          L     1,512(0,13)
0001F4  58 F0 3 094          L     15,A..IBMBSIOT
0001F8  05 EF                BALR  14,15

* STATEMENT NUMBER  44
0001FA  18 0D                LR    0,13
0001FC  58 D0 D 004          L     13,4(0,13)
000200  58 E0 D 00C          L     14,12(0,13)
000204  98 2C D 01C          LM    2,12,28(13)
000208  05 1E                BALR  1,14

* END PROCEDURE
00020A  07 07                NOPR  7

* STATEMENT NUMBER  9

* ON UNIT BLOCK
00020C  90 EC D 00C          STM   14,12,12(13)
000210  47 F0 F 014          B     20(0,15)
000214  00000000             DC    A(STMT. NO. TABLE)
000218  00000110             DC    F'272'
00021C  00000000             DC    A(STATIC CSECT)
000220  58 30 F 010          L     3,16(0,15)
000224  58 10 D 04C          L     1,76(0,13)
000228  58 00 F 00C          L     0,12(0,15)
00022C  1E 01                ALR   0,1
00022E  55 00 C 00C          CL    0,12(0,12)
000232  47 D0 F 030          BNH   48(0,15)
000236  58 F0 C 074          L     15,116(0,12)
00023A  05 EF                BALR  14,15
00023C  58 E0 D 048          L     14,72(0,13)
000240  18 F0                LR    15,0
```

```
000242  90 E0 1 048              STM   14,0,72(1)
000246  50 D0 1 004              ST    13,4(0,1)
00024A  41 D1 0 000              LA    13,0(1,0)
00024E  50 50 D 058              ST    5,88(0,13)
000252  92 8C D 000              MVI   0(13),X'8C'
000256  92 24 D 001              MVI   1(13),X'24'
00025A  58 60 D 058              L     6,88(0,13)
00025E  50 60 D 0C0              ST    6,192(0,13)
000262  D2 03 D 054 3 128        MVC   84(4,13),296(3)
000268  05 20                   BALR  2,0

* PROCEDURE BASE

* STATEMENT NUMBER  10
00026A  1B 11                    SR    1,1
00026C  1B 55                    SR    5,5
00026E  58 F0 3 16C              L     15,364(0,3)
000272  05 EF                   BALR  14,15

* STATEMENT NUMBER  11
000274  D2 03 D 0F0 3 24A        MVC   240(4,13),586(3)
00027A  D2 07 D 0F4 3 100        MVC   244(8,13),256(3)
000280  41 80 D 0F0              LA    8,240(0,13)
000284  50 80 D 0F4              ST    8,244(0,13)
000288  41 90 D 0F4              LA    9,244(0,13)
00028C  50 90 3 170              ST    9,368(0,3)
000290  D2 02 D 0FC 3 24E        MVC   252(3,13),590(3)
000296  D2 07 D 100 3 108        MVC   256(8,13),264(3)
00029C  41 E0 D 0FC              LA    14,252(0,13)
0002A0  50 E0 D 100              ST    14,256(0,13)
0002A4  41 90 D 100              LA    9,256(0,13)
0002A8  50 90 3 174              ST    9,372(0,3)
0002AC  96 80 3 174              OI    372(3),X'80'
0002B0  1B 55                    SR    5,5
0002B2  41 10 3 170              LA    1,368(0,3)
0002B6  58 F0 3 16C              L     15,364(0,3)
0002BA  05 EF                   BALR  14,15

* STATEMENT NUMBER  12
0002BC  D2 03 D 0F0 3 24A        MVC   240(4,13),586(3)
0002C2  D2 07 D 0F4 3 100        MVC   244(8,13),256(3)
0002C8  41 80 D 0F0              LA    8,240(0,13)
0002CC  50 80 D 0F4              ST    8,244(0,13)
0002D0  41 90 D 0F4              LA    9,244(0,13)
0002D4  50 90 3 178              ST    9,376(0,3)
0002D8  D2 06 D 0FC 3 251        MVC   252(7,13),593(3)
0002DE  D2 07 D 104 3 110        MVC   260(8,13),272(3)
0002E4  41 80 D 0FC              LA    8,252(0,13)
0002E8  50 80 D 104              ST    8,260(0,13)
0002EC  41 90 D 104              LA    9,260(0,13)
0002F0  50 90 3 17C              ST    9,380(0,3)
0002F4  96 80 3 17C              OI    380(3),X'80'
0002F8  1B 55                    SR    5,5
0002FA  41 10 3 178              LA    1,376(0,3)
0002FE  58 F0 3 16C              L     15,364(0,3)
000302  05 EF                   BALR  14,15

* STATEMENT NUMBER  13
000304  1B 11                    SR    1,1
000306  1B 55                    SR    5,5
000308  58 F0 3 16C              L     15,364(0,3)
00030C  05 EF                   BALR  14,15

* STATEMENT NUMBER  14
00030E  41 70 6 0D4              LA    7,BINVAR.FBVAR
000312  50 70 3 180              ST    7,384(0,3)
000316  96 80 3 180              OI    384(3),X'80'
00031A  41 10 3 180              LA    1,384(0,3)
00031E  58 F0 3 06C              L     15,A..IBMBEOCA
000322  05 EF                   BALR  14,15

* STATEMENT NUMBER  15
000324  58 40 D 048              L     4,72(0,13)
000328  4A 40 4 002              AH    4,2(0,4)
00032C                   CL.23   EQU   *
00032C  58 40 4 000              L     4,0(0,4)
000330  91 40 4 006              TM    6(4),X'40'
000334  47 80 2 0C2              BZ    CL.23
000338  58 F0 4 010              L     15,16(0,4)
00033C  50 F0 D 0C8              ST    15,200(0,13)
000340  48 70 3 0DA              LH    7,218(0,3)
000344  40 70 6 0F0              STH   7,CVAR
000348  D2 00 6 0F2 F 000        MVC   CVAR+2(1),0(15)

* STATEMENT NUMBER  16
00034E  58 90 D 048              L     9,72(0,13)
000352  4A 90 9 002              AH    9,2(0,9)
000356                   CL.24   EQU   *
000356  58 90 9 000              L     9,0(0,9)
00035A  91 40 9 006              TM    6(9),X'40'
00035E  47 80 2 0EC              BZ    CL.24
000362  58 40 9 018              L     4,24(0,9)
000366  48 80 9 01C              LH    8,28(0,9)
00036A  50 40 D 0D0              ST    4,208(0,13)
00036E  50 80 D 0CC              ST    8,204(0,13)
000372  41 90 0 0FF              LA    9,255(0,0)
000376  19 98                   CR    9,8
```

```
000378  47 D0 2 114              BNH   CL.25
00037C  18 98                    LR    9,8
00037E                   CL.25   EQU   *
00037E  40 90 6 0F0              STH   9,CVAR
000382  4B 90 3 0DA              SH    9,218(0,3)
000386  47 40 2 12E              BM    CL.26
00038A  44 90 2 128              EX    9,CL.27
00038E  47 F0 2 12E              B     CL.28
000392                   CL.27   EQU   *
000392  D2 00 6 0F2 4 000        MVC   CVAR+2(1),0(4)
000398                   CL.26   EQU   *
000398                   CL.28   EQU   *

* STATEMENT NUMBER  17
000398  41 E0 D 0F0              LA    14,240(0,13)
00039C  50 E0 3 184              ST    14,388(0,3)
0003A0  96 80 3 184              OI    388(3),X'80'
0003A4  41 10 3 184              LA    1,388(0,3)
0003A8  58 F0 3 070              L     15,A..IBMBEOLA
0003AC  05 EF                    BALR  14,15
0003AE  58 80 D 0F0              L     8,240(0,13)
0003B2  50 80 D 0D4              ST    8,212(0,13)
0003B6  48 E0 D 0F4              LH    14,244(0,13)
0003BA  50 E0 D 0D8              ST    14,216(0,13)
0003BE  41 90 0 0FF              LA    9,255(0,0)
0003C2  19 9E                    CR    9,14
0003C4  47 D0 2 160              BNH   CL.29
0003C8  18 9E                    LR    9,14
0003CA                   CL.29   EQU   *
0003CA  40 90 6 0F0              STH   9,CVAR
0003CE  4B 90 3 0DA              SH    9,218(0,3)
0003D2  47 40 2 17A              BM    CL.30
0003D6  44 90 2 174              EX    9,CL.31
0003DA  47 F0 2 17A              B     CL.32
0003DE                   CL.31   EQU   *
0003DE  D2 00 6 0F2 8 000        MVC   CVAR+2(1),0(8)
0003E4                   CL.30   EQU   *
0003E4                   CL.32   EQU   *

* STATEMENT NUMBER  18
0003E4  58 40 D 048              L     4,72(0,13)
0003E8  4A 40 4 002              AH    4,2(0,4)
0003EC                   CL.33   EQU   *
0003EC  58 40 4 000              L     4,0(0,4)
0003F0  91 80 4 006              TM    6(4),X'80'
0003F4  47 80 2 182              BZ    CL.33
0003F8  58 E0 4 008              L     14,8(0,4)
0003FC  48 90 4 00C              LH    9,12(0,4)
000400  50 E0 D 0E0              ST    14,224(0,13)
000404  50 90 D 0DC              ST    9,220(0,13)
000408  41 40 0 0FF              LA    4,255(0,0)
00040C  19 49                    CR    4,9
00040E  47 D0 2 1AA              BNH   CL.34
000412  18 49                    LR    4,9
000414                   CL.34   EQU   *
000414  40 40 6 0F0              STH   4,CVAR
000418  4B 40 3 0DA              SH    4,218(0,3)
00041C  47 40 2 1C4              BM    CL.35
000420  44 40 2 1BE              EX    4,CL.36
000424  47 F0 2 1C4              B     CL.37
000428                   CL.36   EQU   *
000428  D2 00 6 0F2 E 000        MVC   CVAR+2(1),0(14)
00042E                   CL.35   EQU   *
00042E                   CL.37   EQU   *

* STATEMENT NUMBER  19
00042E  58 40 D 048              L     4,72(0,13)
000432  4A 40 4 002              AH    4,2(0,4)
000436                   CL.38   EQU   *
000436  58 40 4 000              L     4,0(0,4)
00043A  91 10 4 006              TM    6(4),X'10'
00043E  47 80 2 1CC              BZ    CL.38
000442  58 E0 4 020              L     14,32(0,4)
000446  48 90 4 024              LH    9,36(0,4)
00044A  50 E0 D 0E8              ST    14,232(0,13)
00044E  50 90 D 0E4              ST    9,228(0,13)
000452  41 40 0 0FF              LA    4,255(0,0)
000456  19 49                    CR    4,9
000458  47 D0 2 1F4              BNH   CL.39
00045C  18 49                    LR    4,9
00045E                   CL.39   EQU   *
00045E  40 40 6 0F0              STH   4,CVAR
000462  4B 40 3 0DA              SH    4,218(0,3)
000466  47 40 2 20E              BM    CL.40
00046A  44 40 2 208              EX    4,CL.41
00046E  47 F0 2 20E              B     CL.42
000472                   CL.41   EQU   *
000472  D2 00 6 0F2 E 000        MVC   CVAR+2(1),0(14)
000478                   CL.40   EQU   *
000478                   CL.42   EQU   *

* STATEMENT NUMBER  20
000478  78 00 6 0E0              LE    0,ONCOUNT
00047C  7E 00 3 134              AU    0,308(0,3)
000480  70 00 D 0F0              STE   0,240(0,13)
000484  91 80 D 0F0              TM    240(13),X'80'
000488  48 80 D 0F2              LH    8,242(0,13)
00048C  47 80 2 228              BZ    CL.43
000490  13 88                    LCR   8,8
000492                   CL.43   EQU   *
```

```
000492  40 80 6 0D4          STH   8,BINVAR.FBVAR

* STATEMENT NUMBER  21
000496  18 0D                LR    0,13
000498  58 D0 D 004          L     13,4(0,13)
00049C  58 E0 D 00C          L     14,12(0,13)
0004A0  98 2C D 01C          LM    2,12,28(13)
0004A4  05 1E                BALR  1,14

* ON UNIT BLOCK END
0004A6  07 07                NOPR  7


* STATEMENT NUMBER  22
0004A8                       DC    C'  DUMMY'
0004AF                       DC    AL1(5)

* PROCEDURE                         DUMMY

* REAL ENTRY
0004B0  90 EC D 00C          STM   14,12,12(13)
0004B4  47 F0 F 014          B     *+16
0004B8  00000000             DC    A(STMT. NO. TABLE)
0004BC  000000F0             DC    F'240'
0004C0  00000000             DC    A(STATIC CSECT)
0004C4  58 30 F 010          L     3,16(0,15)
0004C8  58 10 D 04C          L     1,76(0,13)
0004CC  58 00 F 00C          L     0,12(0,15)
0004D0  1E 01                ALR   0,1
0004D2  55 00 C 00C          CL    0,12(0,12)
0004D6  47 D0 F 030          BNH   *+10
0004DA  58 F0 C 074          L     15,116(0,12)
0004DE  05 EF                BALR  14,15
0004E0  58 E0 D 048          L     14,72(0,13)
0004E4  18 F0                LR    15,0
0004E6  90 E0 1 048          STM   14,0,72(1)
0004EA  50 D0 1 004          ST    13,4(0,1)
0004EE  41 D1 0 000          LA    13,0(1,0)
0004F2  50 50 D 058          ST    5,88(0,13)
0004F6  92 80 D 000          MVI   0(13),X'80'
0004FA  92 24 D 001          MVI   1(13),X'24'
0004FE  58 60 D 058          L     6,88(0,13)
000502  50 60 D 0C0          ST    6,192(0,13)
000506  D2 03 D 054 3 128    MVC   84(4,13),296(3)
00050C  05 20                BALR  2,0

* PROCEDURE BASE


* STATEMENT NUMBER  24
00050E  58 70 6 0DC          L     7,PVAR

000512  92 00 7 000          MVI   AVAR,X'00'
000516  58 E0 3 138          L     14,312(0,3)
00051A  50 E0 7 004          ST    14,AVAR+4

* STATEMENT NUMBER  25
00051E  58 40 3 13C          L     4,316(0,3)
000522  50 40 6 0DC          ST    4,PVAR

* STATEMENT NUMBER  26
000526  50 40 D 0B8          ST    4,OVAR

* STATEMENT NUMBER  27
00052A  D2 03 D 0C8 3 258    MVC   200(4,13),600(3)
000530  D2 07 D 0CC 3 100    MVC   204(8,13),256(3)
000536  41 80 D 0C8          LA    8,200(0,13)
00053A  50 80 D 0CC          ST    8,204(0,13)
00053E  41 B0 D 0CC          LA    11,204(0,13)
000542  50 B0 3 188          ST    11,392(0,3)
000546  D2 03 D 0D4 3 25C    MVC   212(4,13),604(3)
00054C  D2 07 D 0D8 3 100    MVC   216(8,13),256(3)
000552  41 90 D 0D4          LA    9,212(0,13)
000556  50 90 D 0D8          ST    9,216(0,13)
00055A  41 B0 D 0D8          LA    11,216(0,13)
00055E  50 B0 3 18C          ST    11,396(0,3)
000562  41 B0 D 0BC          LA    11,SIZE
000566  50 B0 3 190          ST    11,400(0,3)
00056A  41 90 6 0D0          LA    9,BINVAR.RETCODE
00056E  50 90 3 194          ST    9,404(0,3)
000572  96 80 3 194          OI    404(3),X'80'
000576  1B 55                SR    5,5
000578  41 10 3 188          LA    1,392(0,3)
00057C  58 F0 3 198          L     15,408(0,3)
000580  05 EF                BALR  14,15

* STATEMENT NUMBER  28
000582  D2 03 D 0C8 3 258    MVC   200(4,13),600(3)
000588  D2 07 D 0CC 3 100    MVC   204(8,13),256(3)
00058E  41 80 D 0C8          LA    8,200(0,13)
000592  50 80 D 0CC          ST    8,204(0,13)
000596  41 B0 D 0CC          LA    11,204(0,13)
00059A  50 B0 3 19C          ST    11,412(0,3)
00059E  D2 03 D 0D4 3 25C    MVC   212(4,13),604(3)
0005A4  D2 07 D 0D8 3 100    MVC   216(8,13),256(3)
0005AA  41 E0 D 0D4          LA    14,212(0,13)
0005AE  50 E0 D 0D8          ST    14,216(0,13)
0005B2  41 B0 D 0D8          LA    11,216(0,13)
0005B6  50 B0 3 1A0          ST    11,416(0,3)
0005BA  41 B0 D 0BC          LA    11,SIZE
```

```
0005BE  50 B0 3 1A4          ST    11,420(0,3)
0005C2  41 90 6 0D0          LA    9,BINVAR.RETCODE
0005C6  50 90 3 1A8          ST    9,424(0,3)
0005CA  50 D0 D 0E4          ST    13,228(0,13)
0005CE  58 F0 3 020          L     15,32(0,3)
0005D2  50 F0 D 0E0          ST    15,224(0,13)
0005D6  41 B0 D 0E0          LA    11,224(0,13)
0005DA  50 B0 3 1AC          ST    11,428(0,3)
0005DE  96 80 3 1AC          OI    428(3),X'80'
0005E2  1B 55                SR    5,5
0005E4  41 10 3 19C          LA    1,412(0,3)
0005E8  58 F0 3 1B0          L     15,432(0,3)
0005EC  05 EF                BALR  14,15

* STATEMENT NUMBER  29
0005EE  D2 03 D 0C8 3 258    MVC   200(4,13),600(3)
0005F4  D2 07 D 0CC 3 100    MVC   204(8,13),256(3)
0005FA  41 80 D 0C8          LA    8,200(0,13)
0005FE  50 80 D 0CC          ST    8,204(0,13)
000602  41 B0 D 0CC          LA    11,204(0,13)
000606  50 B0 3 1B4          ST    11,436(0,3)
00060A  D2 03 D 0D4 3 25C    MVC   212(4,13),604(3)
000610  D2 07 D 0D8 3 100    MVC   216(8,13),256(3)
000616  41 E0 D 0D4          LA    14,212(0,13)
00061A  50 E0 D 0D8          ST    14,216(0,13)
00061E  41 B0 D 0D8          LA    11,216(0,13)
000622  50 B0 3 1B8          ST    11,440(0,3)
000626  41 B0 D 0BC          LA    11,SIZE
00062A  50 B0 3 1BC          ST    11,444(0,3)
00062E  41 90 6 0D0          LA    9,BINVAR.RETCODE
000632  50 90 3 1C0          ST    9,448(0,3)
000636  50 D0 D 0E4          ST    13,228(0,13)
00063A  58 F0 3 028          L     15,40(0,3)
00063E  50 F0 D 0E0          ST    15,224(0,13)
000642  41 B0 D 0E0          LA    11,224(0,13)
000646  50 B0 3 1C4          ST    11,452(0,3)
00064A  96 80 3 1C4          OI    452(3),X'80'
00064E  1B 55                SR    5,5
000650  41 10 3 1B4          LA    1,436(0,3)
000654  58 F0 3 1C8          L     15,456(0,3)
000658  05 EF                BALR  14,15

* STATEMENT NUMBER  30
00065A  D2 03 D 0C8 3 258    MVC   200(4,13),600(3)
000660  D2 07 D 0CC 3 100    MVC   204(8,13),256(3)
000666  41 80 D 0C8          LA    8,200(0,13)
00066A  50 80 D 0CC          ST    8,204(0,13)
00066E  41 B0 D 0CC          LA    11,204(0,13)
000672  50 B0 3 1CC          ST    11,460(0,3)
000676  D2 03 D 0D4 3 25C    MVC   212(4,13),604(3)
00067C  D2 07 D 0D8 3 100    MVC   216(8,13),256(3)
000682  41 E0 D 0D4          LA    14,212(0,13)
000686  50 E0 D 0D8          ST    14,216(0,13)
00068A  41 B0 D 0D8          LA    11,216(0,13)
00068E  50 B0 3 1D0          ST    11,464(0,3)
000692  41 B0 D 0BC          LA    11,SIZE
000696  50 B0 3 1D4          ST    11,468(0,3)
00069A  41 90 6 0D0          LA    9,BINVAR.RETCODE
00069E  50 90 3 1D8          ST    9,472(0,3)
0006A2  50 D0 D 0E4          ST    13,228(0,13)
0006A6  58 F0 3 020          L     15,32(0,3)
0006AA  50 F0 D 0E0          ST    15,224(0,13)
0006AE  41 B0 D 0E0          LA    11,224(0,13)
0006B2  50 B0 3 1DC          ST    11,476(0,3)
0006B6  50 D0 D 0EC          ST    13,236(0,13)
0006BA  58 F0 3 028          L     15,40(0,3)
0006BE  50 F0 D 0E8          ST    15,232(0,13)
0006C2  41 B0 D 0E8          LA    11,232(0,13)
0006C6  50 B0 3 1E0          ST    11,480(0,3)
0006CA  96 80 3 1E0          OI    480(3),X'80'
0006CE  1B 55                SR    5,5
0006D0  41 10 3 1CC          LA    1,460(0,3)
0006D4  58 F0 3 1E4          L     15,484(0,3)
0006D8  05 EF                BALR  14,15

* STATEMENT NUMBER  31
0006DA  D2 03 D 0C8 3 258    MVC   200(4,13),600(3)
0006E0  D2 07 D 0CC 3 100    MVC   204(8,13),256(3)
0006E6  41 80 D 0C8          LA    8,200(0,13)
0006EA  50 80 D 0CC          ST    8,204(0,13)
0006EE  41 B0 D 0CC          LA    11,204(0,13)
0006F2  50 B0 3 1E8          ST    11,488(0,3)
0006F6  D2 03 D 0D4 3 25C    MVC   212(4,13),604(3)
0006FC  D2 07 D 0D8 3 100    MVC   216(8,13),256(3)
000702  41 E0 D 0D4          LA    14,212(0,13)
000706  50 E0 D 0D8          ST    14,216(0,13)
00070A  41 B0 D 0D8          LA    11,216(0,13)
00070E  50 B0 3 1EC          ST    11,492(0,3)
000712  D2 01 D 0E2 3 260    MVC   226(2,13),608(3)
000718  D2 07 D 0E4 3 118    MVC   228(8,13),280(3)
00071E  41 80 D 0E2          LA    8,226(0,13)
000722  50 80 D 0E4          ST    8,228(0,13)
000726  41 B0 D 0E4          LA    11,228(0,13)
00072A  50 B0 3 1F0          ST    11,496(0,3)
00072E  41 90 6 0D0          LA    9,BINVAR.RETCODE
000732  50 90 3 1F4          ST    9,500(0,3)
000736  96 80 3 1F4          OI    500(3),X'80'
00073A  1B 55                SR    5,5
00073C  41 10 3 1E8          LA    1,488(0,3)
000740  58 F0 3 1F8          L     15,504(0,3)
000744  05 EF                BALR  14,15
```

```
* STATEMENT NUMBER  32
000746  D2 05 D 0CA 3 262        MVC   202(6,13),610(3)
00074C  D2 07 D 0D0 3 0E8        MVC   208(8,13),232(3)
000752  41 E0 D 0CA              LA    14,202(0,13)
000756  50 E0 D 0D0              ST    14,208(0,13)
00075A  41 B0 D 0D0              LA    11,208(0,13)
00075E  50 B0 3 1FC              ST    11,508(0,3)
000762  D2 07 D 0D8 3 120        MVC   216(8,13),288(3)
000768  41 E0 D 0D8              LA    14,216(0,13)
00076C  50 E0 D 0D8              ST    14,216(0,13)
000770  41 B0 D 0D8              LA    11,216(0,13)
000774  50 B0 3 200              ST    11,512(0,3)
000778  D2 01 D 0E2 3 260        MVC   226(2,13),608(3)
00077E  D2 07 D 0E4 3 118        MVC   228(8,13),280(3)
000784  41 80 D 0E2              LA    8,226(0,13)
000788  50 80 D 0E4              ST    8,228(0,13)
00078C  41 B0 D 0E4              LA    11,228(0,13)
000790  50 B0 3 204              ST    11,516(0,3)
000794  41 90 6 0D8              LA    9,CKPT_RETC
000798  50 90 3 208              ST    9,520(0,3)
00079C  96 80 3 208              OI    520(3),X'80'
0007A0  1B 55                    SR    5,5
0007A2  41 10 3 1FC              LA    1,508(0,3)
0007A6  58 F0 3 1F8              L     15,504(0,3)
0007AA  05 EF                    BALR  14,15

* STATEMENT NUMBER  33
0007AC  1B 11                    SR    1,1
0007AE  1B 55                    SR    5,5
0007B0  58 F0 3 20C              L     15,524(0,3)
0007B4  05 EF                    BALR  14,15

* STATEMENT NUMBER  34
0007B6  1B 11                    SR    1,1
0007B8  1B 55                    SR    5,5
0007BA  58 F0 3 210              L     15,528(0,3)
0007BE  05 EF                    BALR  14,15

* STATEMENT NUMBER  35
0007C0  41 90 6 0D0              LA    9,BINVAR.RETCODE
0007C4  50 90 3 214              ST    9,532(0,3)
0007C8  96 80 3 214              OI    532(3),X'80'
0007CC  1B 55                    SR    5,5
0007CE  41 10 3 214              LA    1,532(0,3)
0007D2  58 F0 3 15C              L     15,348(0,3)
0007D6  05 EF                    BALR  14,15

* STATEMENT NUMBER  41
0007D8  18 0D                    LR    0,13
0007DA  58 D0 D 004              L     13,4(0,13)
0007DE  58 E0 D 00C              L     14,12(0,13)
0007E2  98 2C D 01C              LM    2,12,28(13)
0007E6  05 1E                    BALR  1,14

* END PROCEDURE

* STATEMENT NUMBER  36
0007E8                           DC    C'  A'
0007EB                           DC    ALI(1)

* PROCEDURE                             A

* REAL ENTRY
0007EC  90 EC D 00C              STM   14,12,12(13)
0007F0  47 F0 F 014              B     *+16
0007F4  00000000                 DC    A(STMT. NO. TABLE)
0007F8  000000C0                 DC    F'192'
0007FC  00000000                 DC    A(STATIC CSECT)
000800  58 30 F 010              L     3,16(0,15)
000804  58 10 D 04C              L     1,76(0,13)
000808  58 00 F 00C              L     0,12(0,15)
00080C  1E 01                    ALR   0,1
00080E  55 00 C 00C              CL    0,12(0,12)
000812  47 D0 F 030              BNH   *+10
000816  58 F0 C 074              L     15,116(0,12)
00081A  05 EF                    BALR  14,15
00081C  58 E0 D 048              L     14,72(0,13)
000820  18 F0                    LR    15,0
000822  90 E0 1 048              STM   14,0,72(1)
000826  50 D0 1 004              ST    13,4(0,1)
00082A  41 D1 0 000              LA    13,0(1,0)
00082E  50 50 D 058              ST    5,88(0,13)
000832  92 80 D 000              MVI   0(13),X'80'
000836  92 24 D 001              MVI   1(13),X'24'
00083A  D2 03 D 054 3 128        MVC   84(4,13),296(3)
000840  05 20                    BALR  2,0

* PROCEDURE BASE

* STATEMENT NUMBER  37
000842  18 0D                    LR    0,13
000844  58 D0 D 004              L     13,4(0,13)
000848  58 E0 D 00C              L     14,12(0,13)
00084C  98 2C D 01C              LM    2,12,28(13)
000850  05 1E                    BALR  1,14
```

```
* END PROCEDURE
000852  07 07                    NOPR  7

* STATEMENT NUMBER  38
000854                           DC    C'  B'
000857                           DC    AL1(1)

* PROCEDURE                             B

* REAL ENTRY
000858  90 EC D 00C               STM   14,12,12(13)
00085C  47 F0 F 014               B     *+16
000860  00000000                 DC    A(STMT. NO. TABLE)
000864  000000C8                 DC    F'200'
000868  00000000                 DC    A(STATIC CSECT)
00086C  58 30 F 010               L     3,16(0,15)
000870  58 10 D 04C               L     1,76(0,13)
000874  58 00 F 00C               L     0,12(0,15)
000878  1E 01                    ALR   0,1
00087A  55 00 C 00C               CL    0,12(0,12)
00087E  47 D0 F 030               BNH   *+10
000882  58 F0 C 074               L     15,116(0,12)
000886  05 EF                    BA'R  14,15
000888  58 E0 D 048               L     14,72(0,13)
00088C  18 F0                    LR    15,0
00088E  90 E0 1 048               STM   14,0,72(1)
000892  50 D0 1 004               ST    13,4(0,1)
000896  41 D1 0 000               LA    13,0(1,0)
00089A  50 50 D 058               ST    5,88(0,13)
00089E  92 80 D 000               MVI   0(13),X'80'
0008A2  92 24 D 001               MVI   1(13),X'24'
0008A6  D2 03 D 054 3 128         MVC   84(4,13),296(3)
0008AC  58 10 D 004               L     1,4(0,13)
0008B0  58 10 1 018               L     1,24(0,1)
0008B4  D2 03 D 0C0 1 000         MVC   192(4,13),0(1)
0008BA  94 7F D 0C0               NI    192(13),X'7F'
0008BE  05 20                    BALR  2,0

* PROCEDURE BASE

* STATEMENT NUMBER  40
0008C0  18 0D                    LR    0,13
0008C2  58 D0 D 004               L     13,4(0,13)
0008C6  58 E0 D 00C               L     14,12(0,13)
0008CA  98 2C D 01C               LM    2,12,28(13)
0008CE  05 1E                    BALR  1,14

* END PROCEDURE

* END PROGRAM
```

```
PL/I OPTIMIZING COMPILER        /***** PL/I SAMPLE PROGRAM. *****/        PAGE  30

COMPILER DIAGNOSTIC MESSAGES
 (1)  (2)  (3)
ERROR ID L   STMT    MESSAGE DESCRIPTION


SEVERE AND ERROR DIAGNOSTIC MESSAGES

       (4)
IEL0413I E   23      DECLARATION OF INTERNAL ENTRY NOT ALLOWED.    DECLARATION OF 'A' IGNORED.


WARNING DIAGNOSTIC MESSAGES


IEL0892I W   6       TARGET STRING SHORTER THAN SOURCE.    RESULT TRUNCATED ON ASSIGNMENT.

IEL0518I W   20      'ONCOUNT' IS THE NAME OF A BUILTIN FUNCTION BUT ITS IMPLICIT DECLARATION DOES NOT IMPLY
                     'BUILTIN'.


COMPILER INFORMATORY MESSAGES


IEL0533I I           NO 'DECLARE' STATEMENT(S) FOR 'SYSPRINT','PLISRTB','PLIRETC','PLIDUMP','PLICKPT','PLIREST',
                     'PLICANC','PVAR','PLISRTA','ONCOUNT','PLISRTC','PLISRTD'.


END OF COMPILER DIAGNOSTIC MESSAGES
 (5)                              (6)
COMPILE TIME    0.02 MINS        SPILL FILE:     0 RECORDS, SIZE  4051
```

Diagnostic messages and an end of compile step message generated by the compiler. All diagnostic messages generated by the optimizing compiler are documented in the publication OS Optimizing Compiler: Messages.

(1) "ERROR ID". This identifies the message as originating from the optimizing compiler (IEL), and gives the message number.

(2) "L" This is the severity level of the message.

(3) "STMT" This gives the number of the statement in which the error occurred.

(4) The "E" level message is expected and will cause the return code of "8" from the compiler.

(5) Compile time in minutes. This time includes the preprocessor.

(6) This gives the number of records "spilled" into auxiliary storage and the size in bytes of the spill file.

This page intentionally left blank

```
H96-LEVEL LINKAGE EDITOR OPTIONS SPECIFIED XREF,LIST
        DEFAULT OPTION(S) USED -  SIZE=(262144,49152)
```

(1)

(2) CROSS REFERENCE TABLE

(3)

CONTROL SECTION

NAME	ORIGIN	LENGTH
PLISTART	00	50
PLIMAIN	50	8
SYSPINT	58	20
*SAMPLE2	78	36C
IELCGOG	3E8	B6
IELCGOH	4A0	A4
*SAMPLE1	548	8D0
IBMBKCP1*	E18	242
IBMBKST1*	1060	6F8
IBMBPIR1*	1758	45C
IBMBPGR1*	1BB8	666
IBMBPII1*	2220	1418
IBMBPIT1*	3638	268
IBMBXOPT*	38A0	40
IBMBCCC1*	38E0	120
IBMBCCS1*	3A00	19C
IBMBCE01*	3BA0	32E
IBMBCH01*	3ED0	1F2
IBMBCO01*	40C8	44C
IBMBCT01*	4518	2A0
IBMBKDM1*	47B8	108
IBMBPRC1*	48C0	58
IBMBSED1*	4918	558

ENTRY

NAME	LOCATION	NAME	LOCATION	NAME	LOCATION	NAME	LOCATION
PLICALLA	6	PLICALLB	A				
SAMPLE	550						
IBMBKCPA	E18	IBMBKCPB	E1A	IBMBKCPC	E1C		
IBMBKSTA	1060	IBMBKSTB	1062	IBMBKSTC	1064	IBMBKSTD	1066
IBMBPIRA	17A8	IBMBPIRB	17AA	IBMBPIRC	17AC		
IBMBPGRA	1BB8						
IBMBPIIA	2220						
IBMBPITA	3638						
IBMBCCCB	38E0	IBMBCCCC	38E2	IBMBCCCA	38E6		
IBMBCCSA	3A00						
IBMBCEZB	3BA0	IBMBCEDB	3BA8	IBMBCEDX	3BB0	IBMBCEDF	3BB0
IBMBCEFX	3BB0	IBMBCEZF	3BB0	IBMBCEZX	3BB0		
IBMBCHXE	3ED0	IBMBCHFE	3ED0	IBMBCHXP	3ED8	IBMBCHFP	3ED8
IBMBCHXY	3EE0	IBMBCHFY	3EE0	IBMBCHFH	3EE8	IBMBCHXH	3EE8
IBMBCHFD	3EF0	IBMBCHXD	3EF0	IBMBCHXF	3EF8		
IBMBCODE	40C8	IBMBCOZE	40C8	IBMBCODP	40C8		
IBMBCTHD	4518	IBMBCTHZ	4518	IBMBCTHX	4520	IBMBCTHF	4528
IBMBCTHP	4530	IBMBCTHE	4538				
IBMBKDMA	47B8						
IBMBPRCA	48C0						
IBMBSEDA	4928	IBMBSEDB	4928				

See note on next page.

The linkage editor listing.

1. Statement identifying the version and level of the linkage editor and giving the options as specified in the PARM parameter of the EXEC statement that invokes the cataloged procedure.

2. Cross reference table. This table consists of a module map and the cross-reference table.

3. The module map shows each control section and its associated entry points, if any, listed across the page. An asterisk after the name means that these are library subroutines obtained by automatic library call.

4. The cross-reference table gives all the locations in a control section at which a symbol is referenced. $UNRESOLVED (W) identifies a weak external reference that has not been resolved. See next page.

NAME	ORIGIN	LENGTH
IBMBSIO1*	4E70	284
IBMBCU01*	50F8	308
IBMBEOC1*	5400	100
IBMBEOL1*	5500	B0
IBMBJDT1*	55B0	A0
IBMBJTT1*	5650	78
IBMBOCL1*	56C8	1B0
IBMBSAO1*	5878	100
IBMBSEO1*	5978	10C
IBMBSLO1*	5A88	879
IBMBSPL1*	6308	2E0
IBMBCGT1*	65E8	88
IBMBCK01*	6670	19A
IBMBEER1*	6810	4
IBMBERR1*	6818	A7D
IBMBSCV1*	7298	250
IBMBSPO1*	74E8	130
IBMBEEF1*	7618	15D

NAME	LOCATION	NAME	LOCATION	NAME	LOCATION	NAME	LOCATION
IBMBSIOA	4E70	IBMBSIOB	4E72	IBMBSIOC	4E74	IBMBSIOD	4E76
IBMBSIOE	4E78	IBMBSIOT	5052				
IBMBCUIX	50F8	IBMBCUID	5100	IBMBCUIP	5108	IBMBCUIE	5110
IBMBCUIF	5118						
IBMBEOCA	5400						
IBMBEOLA	5500						
IBMBJDTA	55B0						
IBMBJTTA	5650						
IBMBOCLA	56C8	IBMBOCLB	56CA	IBMBOCLC	56CC	IBMBOCLD	56CE
IBMBSAOA	5878						
IBMBSEOA	5978						
IBMBSLOA	5A88	IBMBSLOB	5A8A				
IBMBSPLA	6308	IBMBSPLB	630A	IBMBSPLC	630C		
IBMBCGTA	65E8						
IBMBCKDP	6670	IBMBCKZP	6670	IBMBCKDD	6678	IBMBCKZD	6678
IBMBEERA	6810						
IBMBERRA	6818	IBMBERRB	691E	IBMBERRC	7144		
IBMBSCVA	7298						
IBMBSPOA	74E8						
IBMBEEFA	7618						

LOCATION	REFERS TO SYMBOL	IN CONTROL SECTION	LOCATION	REFERS TO SYMBOL	IN CONTROL SECTION
10	PLIMAIN	(4) PLIMAIN	14	SYSPINT	SYSPINT
18	PLIFLOW	$UNRESOLVED(W)	1C	PLITABS	$UNRESOLVED(W)
2C	PLICOUNT	$UNRESOLVED(W)	30	PLIXOPT	$UNRESOLVED(W)
34	IBMBPOPT	$UNRESOLVED(W)	38	PLIXHD	$UNRESOLVED(W)
3C	IBMBEATA	$UNRESOLVED(W)	44	IBMBPIRA	IBMBPIR1
48	IBMBPIRB	IBMBPIR1	4C	IBMBPIRC	IBMBPIR1
50	*SAMPLE1	*SAMPLE1	7C	*SAMPLE1	*SAMPLE1
80	*SAMPLE1	*SAMPLE1	84	*SAMPLE1	*SAMPLE1
88	*SAMPLE1	*SAMPLE1	8C	*SAMPLE1	*SAMPLE1
90	*SAMPLE1	*SAMPLE1	94	*SAMPLE1	*SAMPLE1

LOCATION	REFERS TO SYMBOL	IN CONTROL SECTION	LOCATION	REFERS TO SYMBOL	IN CONTROL SECTION
98	*SAMPLE1	*SAMPLE1	9C	*SAMPLE1	*SAMPLE1
A0	*SAMPLE1	*SAMPLE1	A4	*SAMPLE1	*SAMPLE1
A8	*SAMPLE1	*SAMPLE1	AC	*SAMPLE1	*SAMPLE1
B0	*SAMPLE1	*SAMPLE1	B4	*SAMPLE1	*SAMPLE1
B8	*SAMPLE1	*SAMPLE1	BC	*SAMPLE1	*SAMPLE1
C0	IELCGOG	IELCGOG	C4	IELCGOH	IELCGOH
C8	IBMBCCCA	IBMBCCC1	CC	IBMBCCSA	IBMBCCS1
D0	IBMBCEDB	IBMBCE01	D4	IBMBCHFD	IBMBCH01
D8	IBMBCODE	IBMBCO01	DC	IBMBCTHD	IBMBCT01
E0	IBMBCUID	IBMBCU01	E4	IBMBEOCA	IBMBEOC1
E8	IBMBEOLA	IBMBEOL1	EC	IBMBJDTA	IBMBJDT1
F0	IBMBJTTA	IBMBJTT1	F4	IBMBOCLA	IBMBOCL1
F8	IBMBOCLC	IBMBOCL1	FC	IBMBSAOA	IBMBSAO1
100	IBMBSEDB	IBMBSED1	104	IBMBSEOA	IBMBSEO1
108	IBMBSIOE	IBMBSIO1	10C	IBMBSIOT	IBMBSIO1
110	IBMBSLOA	IBMBSLO1	114	IBMBSPLA	IBMBSPL1
118	IBMBSPOA	IBMBSPO1	11C	IBMBCKDD	IBMBCK01
1B8	SYSPINT	SYSPINT	1C4	SYSPINT	SYSPINT
1D4	IBMBPRCA	IBMBPRC1	1D8	SYSPINT	SYSPINT
1E4	IBMBKDMA	IBMBKDM1	210	IBMBKSTA	IBMBKST1
228	IBMBKSTB	IBMBKST1	240	IBMBKSTC	IBMBKST1
25C	IBMBKSTD	IBMBKST1	270	IBMBKCPA	IBMBKCP1
284	IBMBKCPB	IBMBKCP1	288	IBMBKCPC	IBMBKCP1
2E4	*SAMPLE1	*SAMPLE1	2EC	*SAMPLE1	*SAMPLE1
320	*SAMPLE1	*SAMPLE1	364	*SAMPLE1	*SAMPLE1
3AC	*SAMPLE1	*SAMPLE1	3C4	*SAMPLE1	*SAMPLE1
53C	IBMBSIST	$UNRESOLVED(W)	540	IBMBSEOA	IBMBSEO1
558	*SAMPLE2	*SAMPLE2	560	*SAMPLE2	*SAMPLE2
75C	*SAMPLE2	*SAMPLE2	764	*SAMPLE2	*SAMPLE2
A00	*SAMPLE2	*SAMPLE2	A08	*SAMPLE2	*SAMPLE2
D3C	*SAMPLE2	*SAMPLE2	D44	*SAMPLE2	*SAMPLE2
DA8	*SAMPLE2	*SAMPLE2	DB0	*SAMPLE2	*SAMPLE2
1750	IBMCKEXA	$UNRESOLVED(W)	1754	IBMCKEXB	$UNRESOLVED(W)
1B28	IBMBJWTA	$UNRESOLVED(W)	1B2C	IBMBTOCA	$UNRESOLVED(W)
1B30	IBMBTOCB	$UNRESOLVED(W)	1B34	IBMBTPRA	$UNRESOLVED(W)
1AD0	IBMBOCLB	IBMBOCL1	1AE4	IBMBOCLB	IBMBOCL1
1B18	IBMBOCLD	IBMBOCL1	1B20	IBMBERRB	IBMBERR1
1B24	IBMBPGOA	$UNRESOLVED(W)	1B38	IBMBPQDA	$UNRESOLVED(W)
1B50	IBMBERRC	IBMBERR1	1AD4	IBMBOCLC	IBMBOCL1
1B68	IBMBPGR1	IBMBPGR1	1B70	IBMBPIIA	IBMBPII1
1B74	IBMBPITA	IBMBPIT1	1AD8	IBMBOCLA	IBMBOCL1
1B14	IBMBOCLA	IBMBOCL1	1B1C	IBMBERRA	IBMBERR1
1B58	IBMBEERA	IBMBEER1	303C	IBMBPIRA	IBMBPIR1
3040	IBMBXOPT	IBMBXOPT	3AD0	IBMBCHXD	IBMBCH01
3AD4	IBMBCHXF	IBMBCH01	3AE0	IBMBCHXP	IBMBCH01
3AE4	IBMBCHXY	IBMBCH01	3AE8	IBMBCHXE	IBMBCH01
3B10	IBMBCKDD	IBMBCK01	3B20	IBMBCKDP	IBMBCK01
3B50	IBMBCHFD	IBMBCH01	3B5C	IBMBCHFH	IBMBCH01
3B60	IBMBCHFP	IBMBCH01	3B64	IBMBCHFY	IBMBCH01
3B68	IBMBCHFE	IBMBCH01	3B0C	IBMBCEDX	IBMBCE01
3B14	IBMBCEDF	IBMBCE01	3B4C	IBMBCEFX	IBMBCE01
3ACC	IBMBCYXX	$UNRESOLVED(W)	3B54	IBMBCYFF	$UNRESOLVED(W)

LOCATION	REFERS TO SYMBOL	IN CONTROL SECTION	LOCATION	REFERS TO SYMBOL	IN CONTROL SECTION
3B8C	IBMBCMPX	$UNRESOLVED(W)	3B90	IBMBCMPD	$UNRESOLVED(W)
3B94	IBMBCMPF	$UNRESOLVED(W)	3B80	IBMBCUIX	IBMBCU01
3B88	IBMBCUIF	IBMBCU01	3B40	IBMBCTHX	IBMBCT01
3B44	IBMBCTHD	IBMBCT01	3B48	IBMBCTHF	IBMBCT01
3B28	IBMBCODE	IBMBCO01	3B24	IBMBCVDY	$UNRESOLVED(W)
3AF0	IBMBCRXB	$UNRESOLVED(W)	3AF8	IBMBCRXB	$UNRESOLVED(W)
3B70	IBMBCRXB	$UNRESOLVED(W)	3B78	IBMBCRXB	$UNRESOLVED(W)
3B1C	IBMBCWDH	$UNRESOLVED(W)	3B04	IBMBCGZA	$UNRESOLVED(W)
3B08	IBMBCGPA	$UNRESOLVED(W)	3AEC	IBMBCACA	$UNRESOLVED(W)
3AF4	IBMBCACA	$UNRESOLVED(W)	3AFC	IBMBCACA	$UNRESOLVED(W)
3B2C	IBMBCACA	$UNRESOLVED(W)	3B34	IBMBCACA	$UNRESOLVED(W)
3B3C	IBMBCACA	$UNRESOLVED(W)	3B6C	IBMBCACA	$UNRESOLVED(W)
3B74	IBMBCACA	$UNRESOLVED(W)	3B7C	IBMBCACA	$UNRESOLVED(W)
3B98	IBMBCPBF	$UNRESOLVED(W)	3ADC	IBMBCHXH	IBMBCH01
3B30	IBMBCEDB	IBMBCE01	3B38	IBMBCEDB	IBMBCE01
3B00	IBMBSCVA	IBMBSCV1	3BA4	IBMBCRFB	$UNRESOLVED(W)
3BAC	IBMBCRXB	$UNRESOLVED(W)	3EC4	IBMBCGTA	IBMBCGT1
3ED4	IBMBCOZE	IBMBCO01	3EDC	IBMBCKZP	IBMBCK01
3EE4	IBMBCVZY	$UNRESOLVED(W)	3EEC	IBMBCWZH	$UNRESOLVED(W)
3EF4	IBMBCKZD	IBMBCK01	40A4	IBMBCGTA	IBMBCGT1
451C	IBMBCKZD	IBMBCK01	453C	IBMBCOZE	IBMBCO01
452C	IBMBCEZF	IBMBCE01	4534	IBMBCKZP	IBMBCK01
4524	IBMBCEZX	IBMBCE01	47B4	IBMBSCVA	IBMBSCV1
4E54	IBMBSAIA	$UNRESOLVED(W)	4E64	IBMBSAOA	IBMBSAO1
4E4C	IBMBSFIA	$UNRESOLVED(W)	4E5C	IBMBSFOA	$UNRESOLVED(W)
4E50	IBMBSPIA	$UNRESOLVED(W)	4E60	IBMBSPOA	IBMBSPO1
4E58	IBMBSCIA	$UNRESOLVED(W)	4E68	IBMBSCOA	$UNRESOLVED(W)
4E6C	IBMBSDOA	$UNRESOLVED(W)	4E48	IBMBSIST	$UNRESOLVED(W)
50E8	IBMBSPLA	IBMBSPL1	50EC	IBMBSPLB	IBMBSPL1
50F0	IBMBSPLC	IBMBSPL1	50E4	IBMBOCLA	IBMBOCL1
50FC	IBMBCEFX	IBMBCE01	5104	IBMBCHFD	IBMBCH01
510C	IBMBCHFP	IBMBCH01	5114	IBMBCHFE	IBMBCH01
53F4	IBMBSCVA	IBMBSCV1	5860	IBMBSCPA	$UNRESOLVED(W)
5964	IBMBCCCA	IBMBCCC1	5968	IBMBCBCA	$UNRESOLVED(W)
596C	IBMBCACA	$UNRESOLVED(W)	5970	IBMBCXOA	$UNRESOLVED(W)
5974	IBMBCGGB	$UNRESOLVED(W)	5A80	IBMBSIST	$UNRESOLVED(W)
62F4	IBMBCACA	$UNRESOLVED(W)	62F0	IBMBCBCA	$UNRESOLVED(W)
62EC	IBMBCZCA	$UNRESOLVED(W)	62E8	IBMBCXOA	$UNRESOLVED(W)
62E4	IBMBSIST	$UNRESOLVED(W)	62F8	IBMXGRAP	$UNRESOLVED(W)
6674	IBMBCODP	IBMBCO01	7254	IBMBERCA	$UNRESOLVED(W)
7258	IBMBEEFA	IBMBEEF1	75D8	IBMBCHXP	IBMBCH01
75DC	IBMBCKDP	IBMBCK01	75E0	IBMBCHFP	IBMBCH01
75EC	IBMBCMPP	$UNRESOLVED(W)	75F4	IBMBCMPP	$UNRESOLVED(W)
75F8	IBMBCHXE	IBMBCH01	75FC	IBMBCODE	IBMBCO01
7600	IBMBCHFE	IBMBCH01	760C	IBMBCMPE	$UNRESOLVED(W)
7614	IBMBCMPE	$UNRESOLVED(W)	7604	IBMBCPBP	$UNRESOLVED(W)
7608	IBMBCPBE	$UNRESOLVED(W)	7610	IBMBCCSA	IBMBCCS1

LOCATION 20 REQUESTS CUMULATIVE PSEUDO REGISTER LENGTH
ENTRY ADDRESS 00

TOTAL LENGTH 7778

```
****GO        DOES NOT EXIST BUT HAS BEEN ADDED TO DATA SET    AMODE 31
RMODE IS ANY
AUTHORIZATION CODE IS         0.
```

```
SAMPLE PROGRAM: DATE = 85/05/01, TIME = 14.24.09
END SAMPLE PROGRAM
```

Execution of the sample program produces a return code of 0101.

APPENDIX E. USING THE OS PL/I OPTIMIZER UNDER VM/PC

This section describes how to use the OS PL/I Optimizing Compiler under Virtual Machine/Personal Computer (VM/PC).

VM/PC is an IBM licensed program that runs on the IBM XT/370 Personal Computer, as an IBM Personal Computer Disk Operating System application. VM/PC gives you an interactive system that has the characteristics of a VM/SP Release 2 system: command entry, command formats, messages, screen formats, file naming conventions, key functions, and application interfaces.

To use the OS PL/I Optimizing Compiler under VM/PC, a host system must be available; this is because you must copy (download) PL/I from the host system into your local VM/PC storage. After you have done this, you can use PL/I either independently of the host system, or in connection with the host system.

VM/PC lets you set up a local System/370 environment in which to do your work, known as a local session. After you have downloaded PL/I into your local storage, you can use it in local sessions.

VM/PC also lets you set up an IBM 3277 or 3101 terminal connection with a host system on a remote computer, so that your personal computer acts as a terminal on the host system; such a connection is known as a remote session. You can use the product in remote sessions as well as in local sessions.

To develop OS PL/I optimizing programs with VM/PC, you'll use both types of sessions. For further details on VM/PC see Virtual Machine/Personal Computer User's Guide Supplement for System/370 Languages.

METHODS OF USING THE OS PL/I OPTIMIZER UNDER VM/PC

There are two different methods you follow to use the OS PL/I Optimizer under VM/PC:

1. Copy (download) the OS PL/I Optimizer modules onto local disk files and then invoke the OS PL/I Optimizer in local sessions. (You need download only when you first access the OS PL/I Optimizer, or when a new release has been installed on the host system.)

2. Link to the host-system minidisk containing the OS PL/I Optimizer and then access it from the local session as a remote minidisk. (You must do this after every Initial Program Load (IPL) of CMS, and whenever the link to the host system fails.)

   Depending on your link with the system and on the system load, this often is not an efficient way to operate. Therefore, it is not further described in this book.

DOWNLOADING THE OS PL/I OPTIMIZER INTO VM/PC

To use the OS PL/I Optimizer under VM/PC in local sessions, you can copy (download) certain OS PL/I Optimizer modules into your local files. The modules you must copy are listed in Figure 161 on page 448.

```
CMSLIB   TXTLIB   (needed for execution only)
IEAXPALL MODULE   (extended precision module)
IEAXPDXR MODULE   (extended precision module)
IEAXPSIM MODULE   (extended precision module)
IEL0AC   MODULE
IEL0AE   MODULE
IEL0AI   MODULE
IEL0AT   MODULE
IEL0BA   MODULE
IEL0CA   MODULE
IEL0CA1  MODULE
IEL0CB   MODULE
IEL0CC   MODULE
IEL0CE   MODULE
IEL0CI   MODULE
IEL0EA   MODULE
IEL0EC   MODULE
IEL0EE   MODULE
IEL0EI   MODULE
IEL0GA   MODULE
IEL0GE   MODULE
IEL0GI   MODULE
IEL0GM   MODULE
IEL0IA   MODULE
IEL0ID   MODULE
IEL0IE   MODULE
IEL0II   MODULE
IEL0IK   MODULE
IEL0IM   MODULE
IEL0IQ   MODULE
IEL0KA   MODULE
IEL0KE   MODULE
IEL0KI   MODULE
IEL0KK   MODULE
IEL0KM   MODULE
IEL0KQ   MODULE
IEL0KT   MODULE
IEL0KV   MODULE
IEL0KX   MODULE
IEL0OA   MODULE
IEL0OC   MODULE
IEL0OE   MODULE
IEL0OI   MODULE
IEL0OM   MODULE
IEL0OX   MODULE
```

Figure 161 (Part 1 of 2). OS PL/I Optimizer Modules Needed for Downloading

```
IELOPA   MODULE
IELOPAL1 MODULE
IELOPAL2 MODULE
IELOPAL3 MODULE
IELOPAL4 MODULE
IELOPAL5 MODULE
IELOPC   MODULE
IELOPE   MODULE
IELOPI   MODULE
IELOQA   MODULE
IELOQE   MODULE
IELOSA   MODULE
IELOSC   MODULE
IELOSD   MODULE
IELOSI   MODULE
IELOSK   MODULE
IELOSM   MODULE
IELOSQ   MODULE
IELOQI   MODULE
IELOUA   MODULE
IELOUAC  MODULE
IELOUAE  MODULE
IELOUAS  MODULE
IELOUAT  MODULE
IELOUAY  MODULE
IELOUE   MODULE
PLILIB   TXTLIB   (needed for execution only)
PLIOPT   MODULE
```

Figure 161 (Part 2 of 2). OS PL/I Optimizer Modules Needed for Downloading

Downloading is necessary only when you first access the OS PL/I Optimizer, or after a new release has been installed on the host system.

In your local VM/PC system, in order to avoid the following warning message:

```
DMS PLI074W ERROR RESETTING AUXILIARY DIRECTORY
```

you should make sure that the R disk is the minidisk containing the OS PL/I Optimizer. Figure 162 on page 451 shows you the commands you must issue.

The procedure is as follows:

1. Link (if necessary) and access the local minidisk that is the target minidisk for the copy operation. If the target minidisk is your own minidisk, the link is not required.
2. Link and access the host minidisk that contains the OS PL/I Optimizer modules.
3. Copy the OS PL/I Optimizer modules from the host minidisk to the local minidisk. (This is known as downloading.)
4. Release the host OS PL/I Optimizer minidisk; it is no longer required.
5. If you are compiling AND executing, link and access the host minidisk that contains the file 'CMSLIB TXTLIB', if not previously downloaded.
6. Copy 'CMSLIB TXTLIB' (if not previously downloaded) to the local minidisk.
7. Release the host CMS minidisk; it is no longer required.

VIRTUAL STORAGE REQUIREMENTS: Approximately 1.0M bytes

MINIDISK STORAGE REQUIREMENTS: Approximately 3.0M bytes

Notes:

1. These storage requirements are for the OS PL/I Optimizer compiler and library only; additional storage is needed for the source and/or object program files.

2. Files PLILIB TXTLIB and CMSLIB TXTLIB take approximately one third of the space requirement given above.

   If you are going to do only compilations under VM/PC, these two files may be omitted.

   If you are going to do both compilations and executions under VM/PC, all of the files listed in Figure 160 on page 448, except for the optional extended precision modules, are required.

INVOKING THE OS PL/I OPTIMIZER UNDER VM/PC

You must first make the OS PL/I Optimizer available on a minidisk you can access. For example:

```
CP LINK vm/pc-id ttt aaa RR read-password
ACCESS aaa R
GLOBAL TXTLIB PLILIB CMSLIB
```

Because you must issue these commands each time you log on to VM/PC, it's a good idea to put the commands you need in your profile EXEC.

Next, you can invoke the OS PL/I Optimizer through the following command:

```
PLIOPT PLISAMP
```

where PLISAMP is the name of your source program. (Its filetype is PLIOPT or PLI.)

You can also specify compiler options. For example:

```
PLIOPT PLISAMP (options
```

which allows you to modify the default compiler options in force for your organization.

```
*************************************************************************
*
* 1)  Link and access the target minidisk.
*
CP LINK vm/pc-id ttt aaa W write-password
ACCESS aaa filemodel
*
* 2) Link and access the host minidisk that contains the OS PL/I
*     Optimizer modules.
*
CP LINK host-id hhh bbb RR read-password REMOTE
ACCESS bbb filemode2
*
* 3) Copy the files you need.
*
COPYFILE filename filetype filemode2 = = filemodel
*
* 4) Release the OS PL/I Optimizer host minidisk.
*
RELEASE filemode2 (DET
*
* 5) Link and access the host minidisk that contains CMSLIB TXTLIB
*     (if not previously downloaded).
*
CP LINK host-id ccc bbb RR read-password REMOTE
ACCESS bbb filemode3
*
* 6) Copy CMSLIB TXTLIB (if not previously downloaded) to the local
VM/PC target minidisk.
*
COPYFILE CMSLIB TXTLIB filemode3 = = filemodel
*
* 7) Release the CMS host minidisk.
*
RELEASE filemode3 (DET
*
*
* Where:
*     ttt - is the virtual address of the local target minidisk that
*           will store the OS PL/I Optimizer modules.
*     aaa - is an unused virtual address on the local VM/PC machine.
*     hhh - is the virtual address of the host minidisk that contains
*           the OS PL/I Optimizer modules.
*     bbb - is the virtual disk address you use to refer to the host disk.
*     ccc - is the virtual address of the host minidisk that contains
*           CMSLIB TXTLIB.
*     filemodel - is the filemode of the local VM machine. To avoid a
*     warning message, specify filemode R.
*     filemode2 - is the filemode of the host minidisk that contains
*           the OS PL/I Optimizer modules.
*     filemode3 - is the filemode of the host CMS TXTLIB minidisk.
*
*************************************************************************
```

Figure 162. CMS Commands to Download the OS PL/I Optimizer

OS PL/I OPTIMIZER PROGRAMMING TIPS

You can improve processing time if you specify OS PL/I Optimizer compiler options that do not request printed listings: NOAGGREGATE, NOATTRIBUTES, NOESD, NOINSOURCE, NOLIST, NOMAP, NOOPTIONS, NOSOURCE, NOSTORAGE, and NOXREF.

OS PL/I OPTIMIZER RESTRICTIONS

The following processing capabilities are not available when you are executing an object program in a local session:

- VSAM file processing is not available.
- Magnetic tape processing is not available.
- The Graphical Data Display Manager (GDDM) is not available.

APPENDIX F. MVS/EXTENDED ARCHITECTURE (MVS/XA) CONSIDERATIONS

This section serves as a general introduction to MVS/XA and its facilities. For more specific details, see MVS/XA Conversion Notebook and MVS/XA System Programming Library: 31-Bit Addressing. The content of those books, with which the reader should be familiar, is not repeated here. Readers who understand MVS/XA and such concepts as the MVS/XA machine's current addressing mode (AMODE) and the AMODE and RMODE attributes of load modules may wish to skip to the section entitled "Use of MVS/XA Facilities by PL/I Release 5" on page 457.

System/370 Extended Architecture (sometimes shortened to 370/XA) represents a set of changes and enhancements to the architecture of the IBM System/370.

Pre-Release 5 PL/I object modules, pre-Release 5 PL/I library modules, and the PL/I Checkout Compiler's interpreter facilities all contain instruction sequences that are incompatible with 31-bit addressing on MVS/XA, but they are supported on MVS/XA in 24-bit addressing mode so that you can continue to use them.

If only Release 5 object modules and resident libraries are linked together, the resulting load module is RMODE(ANY) and AMODE(31), and is compatible with 31-bit addressing on MVS/XA. This load module is also compatible with MVS/SP 1.3 execution.

SYSTEM 370 AND 370/XA DIFFERENCES

The areas of difference between System/370 and 370/XA are as follows:

1. 370/XA permits virtual storage addresses to be 31-bit binary numbers, whereas System/370 treated them as 24-bit binary numbers.

   This fact is of direct importance to you as a PL/I programmer. The storage which your program "sees" and for which you code your program is virtual storage. PL/I Release 5 offers you ways to control and utilize the capability of 370/XA to address more virtual storage by the use of 31-bit addresses.

   While the largest storage address which can be utilized on System/370 is 16 megabytes, the largest such address on 370/XA is 128 times greater: 2 gigabytes, or 2048 megabytes.

   Most of the material in this Appendix and all of the MVS/XA support in PL/I Release 5 are related to 31-bit addressing of virtual storage on MVS/XA.

2. 370/XA permits real storage addresses to be 31-bit binary numbers, whereas System/370 treated them as 24-bit binary numbers (or for certain purposes on certain processor models, 25-bit or 26-bit binary numbers).

   This fact is only of indirect importance to you as a PL/I programmer. The storage your program "sees" is virtual storage. Your PL/I program cannot "see" real storage or affect the way real storage is used to implement virtual storage for your program.

   The amount of real storage available to support the execution of your program may, of course, impact your program's performance by determining the amount of paging required to execute your program.

3. 370/XA provides a new interface between a central processor and the I/O devices attached to it. This interface, called the Dynamic Channel Subsystem, supersedes the channel architecture defined by System/370.

   This facility is only of indirect importance to you as a PL/I programmer. Your program probably performs I/O operations, but the details of how this is done are handled by the hardware and the operating system, not by your PL/I program.

   Your specification of a PL/I file declaration or a PL/I environment option is not affected by the Dynamic Channel Subsystem.

4. 370/XA provides a facility called the Interpretive Execution Facility which supersedes the coding techniques and hardware assists employed on System/370 to support "virtual machines". This facility of 370/XA is supported by the VM/XA Migration Aid.

   This facility is of interest only indirectly to you as a PL/I programmer. Your program might be executed under MVS or MVS/XA in a virtual machine provided by the 370/XA Interpretive Execution Facility and the VM/XA Migration Aid, but you would not be able to control or influence this fact, and nothing you coded in your PL/I program would depend on this fact.

The rest of this appendix will be devoted to describing the 31-bit addressing of virtual storage, and how this relates to your PL/I program under PL/I Release 5.

COMPATIBILITY CONSIDERATIONS

Pre-Release 5 PL/I object modules, pre-Release 5 PL/I library modules, and the PL/I Checkout Compiler's interpreter facilities all contain instruction sequences that are incompatible with 31-bit addressing on MVS/XA, but they are supported on MVS/XA in 24-bit addressing mode so that you can continue to use them.

The fact that a computer implements 370/XA architecture and is being controlled by the MVS/XA operating system does not automatically mean that any program written for System/370 can run on that computer and obtain the benefits of 31-bit addressing of virtual storage.

In fact, many, if not most, programs written for System/370 contain instruction sequences that are incompatible with 370/XA. The most common area of difficulty (though not the only one) relates to the use of the high-order byte of a binary word which is to be used as a storage address to hold some unrelated entity. This was a common and appropriate programming technique when storage was a much scarcer resource than it is today.

Very few of System/370 application programs running today are suitable for exploitation of 31-bit addressing. Since these programs must continue to function, however, 370/XA was equipped with the capability to operate in "24-bit addressing mode".

Practically all System/370 programs that run on MVS (or VS1) today can run on MVS/XA in 24-bit addressing mode. In particular, this includes your PL/I programs that have been compiled by Release 5 of the PL/I Optimizing Compiler or prior releases or by Release 3 of the PL/I Checkout Compiler.

At any one time, a computer which is executing in 370/XA mode is using either 31-bit addressing mode or 24-bit addressing mode. This addressing mode is implemented by generating storage addresses from the instructions that comprise a program and then using either the last 31 bits or the last 24 bits as the effective generated address.

Which interpretation, (that is, which addressing mode) is used depends on a bit (the "AMODE" bit) in a control word of the computer called the Program Status Word or PSW.

When the addressing mode is 24-bit, the program cannot address data or branch to an instruction at any address higher than the largest 24-bit binary number: 16M or 16,777,215 bytes. This address value represents an addressing boundary that a program cannot cross as long as the current addressing mode remains 24-bit.

When the addressing mode is 31-bit, the program can address data or branch to an instruction at any address up to the largest 31-bit binary number: 2 gigabytes, 2048 megabytes, or 2,147,483,647 bytes. A program which is executing with 31-bit as the current addressing mode can reference data or branch to an instruction at any address up to the above limit, including, of course, data and instructions located at addresses below 16 megabytes.

370/XA supplies instructions which can be executed by any program to switch the setting of the AMODE bit, and thus switch the machine to a different addressing mode. This process is called mode switching, and the mode of operation it permits, that of alternating between 24-bit addressing mode and 31-bit addressing mode, is called "bi-modal operation". Both modes are entirely native modes of operation for a 370/XA processor. The MVS/XA operating system itself uses both of these modes of operation for different portions of its own code.

CONSIDERATIONS FOR RELEASE 4 PROGRAMS

When you execute one of your old programs on MVS/XA, for example one compiled by Release 4 of the PL/I Optimizing Compiler, there are instruction sequences embedded within it which cannot work correctly in 31-bit addressing mode. Therefore, the MVS/XA operating system must set the addressing mode to 24-bit before it calls your program.

For your program to be able to access its own instructions and data, in fact, MVS/XA must have loaded your program into storage below 16 megabytes. These facts imply that MVS/XA must have been able to tell ahead of time that your program was restricted to operate in 24-bit addressing mode.

MVS/XA addresses this requirement by assigning attributes to load modules. These attributes define two things:

- Where MVS/XA is to place the load module when it loads it into storage. This property is called "residency mode", or "RMODE".
- What addressing mode MVS/XA is to establish in the computer before it branches to the load module. This attribute of a load module is called the "addressing mode", or "AMODE".

AMODE in this context is a load module attribute, not the current addressing mode of the computer. RMODE and AMODE are simply attributes given to load modules by the linkage editor and honored by the operating system when it loads the load module into storage and branches to it. If the load module thereafter switches to a different AMODE, that is its business, presuming that it is coded correctly to accomplish what it needs to do.

A PL/I Release 4 load module has by default the attributes RMODE(24) and AMODE(24). Thus MVS/XA knows to load it into storage below 16 megabytes and knows to establish 24-bit addressing mode as the current addressing mode before branching to it.

These default values of AMODE 24 and RMODE 24 apply in general unless specific action is taken to override them, and they should not be overridden unless it is certain that a particular load module is capable of executing successfully in AMODE 31.

AMODE RMODE EXCEPTIONS TO DEFAULTS

If only Release 5 object modules and resident libraries are linked together, the resulting load module is AMODE(31) and RMODE(ANY) and is compatible with 31-bit addressing on MVS/XA. This load module is also compatible with MVS/SP 1.3 execution.

Other settings of AMODE and RMODE can arise in two different ways:

1. A language processor other than the PL/I Release 5 compiler may introduce output object modules flagged with an indication such as AMODE(ANY) and RMODE(ANY). The linkage editor, when it finds that all such load modules or object modules that it combines into a load module have these attributes, will assign AMODE(31) and RMODE(ANY) to the resultant load module. A single RMODE(24) suffices to "demote" the resultant load module to RMODE(24). An object module flagged AMODE(24) produces an entry point with the attributes AMODE(24) and RMODE(24).

   Note: Early versions of the operating system assigned an AMODE corresponding to the entry.

2. You can supply linkage editor parameter specifications to force the linkage editor to supply specific attributes to a load module. This is safely and commonly done to force either RMODE(24) or the combination of RMODE(24) and AMODE(24).

   If you use this facility to force the linkage editor to assign attributes indicating that 31-bit addressing mode is acceptable when this is not what the object module information supplied to the linkage editor would normally imply, it is your responsibility to establish that the program can actually be executed with your specifications.

AMODE AND RMODE SUMMARY

Summarizing somewhat more precisely the AMODE and RMODE attributes of load modules:

- AMODE can be set to any of the following values:
  - AMODE 24. The program is designed to receive control in 24-bit addressing mode.
  - AMODE 31. The program is designed to receive control in 31-bit addressing mode.
  - AMODE ANY. The program is designed to receive control in either addressing mode.
- RMODE can be set to any of the following values:
  - RMODE 24. The program is design to reside below 16 megabytes in virtual storage. MVS/XA will always place the program below 16 megabutes.
  - RMODE ANY. The program is designed to reside at any virtual storage location, either above or below 16 megabytes. MVS/XA places the program above 16 megabytes unless there is no suitable virtual storage above 16 megabytes.

USE OF MVS/XA FACILITIES BY PL/I RELEASE 5

The MVS/XA concepts of current addressing mode and the load module attributes of AMODE and RMODE imply the following general characteristics of program execution under MVS/XA.

1. If a program is ever to execute in 24-bit addressing mode, it must reside below 16 megabytes. A load module containing such a program must be given the RMODE(24) attribute to cause MVS/XA to load it into storage below 16 megabytes.

2. If a program is to refer to data located at addresses above 16 megabytes, then it must at that moment be executing in 31-bit addressing mode.

3. Since the implementation mechanism used by overlay load modules is incompatible with 31-bit addressing, MVS/XA supports overlay modules in, and only in, 24-bit addressing mode.

CHARACTERISTICS OF RELEASE 5 MODULES

Object modules and load modules produced by PL/I Release 5 and the PL/I Resident Library Release 5 have the following specific characteristics:

1. Object modules produced by PL/I Release 5 have the attributes AMODE(ANY) and RMODE(ANY). The library modules that comprise the Release 5 PL/I Resident Library also have the attributes AMODE(ANY) and RMODE(ANY). As a result, a load module comprised entirely of PL/I Release 5 object code and PL/I Release 5 Resident Library code will be given by default the attributes AMODE(31) and RMODE(ANY) by the MVS/XA linkage editor. Such load modules can be loaded into storage above 16 megabytes and can be executed entirely in 31-bit addressing mode. In this case, PL/I STATIC storage, since it is link edited with the load module, also resides above 16 megabytes.

   You can override the default load module attributes when you link edit your load module if you wish to do so.

2. PL/I Release 5 load modules may utilize the HEAP option to separate CONTROLLED variables and dynamically allocated BASED variables from the storage associated with the ISA. If the HEAP option is used and the PL/I Release 5 program is being executed in 31-bit addressing mode, then PL/I variables allocated with a PL/I ALLOCATE statement may reside above 16 megabytes.

3. The control blocks that comprise the PL/I execution-time environment and the PL/I execution-time stack, including all AUTOMATIC storage, reside below 16 megabytes, regardless of the program's AMODE and regardless of whether or not the HEAP option is used.

ASSEMBLER ROUTINE TO MODE-SWITCH

No mode-switching between PL/I programs link edited into a single load module is provided by PL/I Release 5. You may insert calls to an Assembler Language program to switch modes provided that you conform to the general MVS/XA constraints described above.

An example of a linkage assist routine to switch from a caller's 31-bit or 24-bit addressing mode to 24-bit addressing mode and back upon return appears in Figure 163 on page 458. The example shows only the code necessary for mode-switching. The following are not shown in the figure:

1. register saving

2. save area chaining

3. acquiring a PL/I DSA or a save area

Figure 127 on page 297 gives an example of this omitted code.

A PL/I program would call this linkage assist routine to transfer control to an AMODE(24) routine in the same load module. The linkage assist routine would switch to 24-bit addressing mode and transfer control to the AMODE(24) routine. The AMODE(24) routine would return control to the linkage assist routine, which would switch back to the caller's addressing mode before returning control to the caller.

Variations of the code sequence in Figure 163 can be used to switch modes within a PL/I procedure or to switch from 24-bit or 31-bit addressing mode to 31-bit addressing mode and back. To switch modes within a PL/I procedure, a linkage assist routine could switch addressing mode and return to the caller, allowing the caller to process in a different addressing mode. A similar linkage assist routine could be called to return to the original addressing mode. When switching to 31-bit addressing mode, the leftmost bit of the target address register must be on before the BASSM or BSM instruction is executed.

```
GLUEMOD  CSECT ,
GLUEMOD  AMODE ANY
GLUEMOD  RMODE 24
         USING *,R15          now executing AMODE 31 or 24
*  Testing the CVT for MVS/XA is necessary only if the
*  program must be portable between XA and non-XA systems
         ST    Rn,SAVERn      save caller's register content
*                             (use a DSA for reentrant code)
         L     Rn,16(0,0)     locate CVT
         USING CVT,Rn         addressability for CVT
         TM    CVTDCB,CVTMVSE  are we on MVS/XA ?
         BO    MVSXA          yes, go switch AMODE
         L     Rn,SAVERn      restore caller's register content
         L     R15,=V(COBOLRTN) call COBOL
         BR    R15            branch to COBOL routine
*                             and COBOL will go back to PL/I
MVSXA    DS    0H
         L     Rn,SAVERn      restore caller's register content
         ST    R14,SAVER14    save caller's return addr
*                             some place (a DSA is needed for
*                             reentrant code)
         L     R15,=V(COBOLRTN) call COBOL routine
         BASSM R14,R15        switching to AMODE(24) and
*                             saving old AMODE
*  need to reestablish base register when control returns
         DROP  R15
         USING *,R14
         L     R14,SAVER14    Return address
         BSM   0,R14          Return to caller restoring
*                             caller's AMODE
         CVT   DSECT=YES
```

Note: BASSM and BSM are valid only on MVS/XA machines. The program executing a mode switch must be RMODE(24) unless the BASSM or BSM target instruction is in an RMODE(24) module.

Reference documents:

- MVS/XA Conversion Notebook - GC28-1143
- MVS/XA System Programming Library:
  31-Bit Addressing - GC28-1158

Figure 163. Example of Code for Mode-Switching

BIT DATA TYPE RESTRICTION

User variables with the BIT data type must reside at virtual storage addresses below 256 megabytes. This requirement arises because certain code sequences generated by the PL/I Optimizing Compiler form "bit-level" data addresses. Such an address is obtained by generating in a general purpose register of the computer the address of the byte of storage which contains the first bit of the bit string or bit array and then multiplying by 8 (shifting that address left three bits) and inserting the bit offset of the first bit.

If the byte address was a 31-bit address, then the resultant bit-level address would be 34 bits long. Such an address will not fit within the 32-bit general purpose registers of System/370 or 370/XA. This problem will not arise if the current addressing mode is 24-bit or if the byte address of the bit variable contains no more than 28 significant bits (256 meg).

If this problem arises at execution time and either the PL/I SPIE or STAE option is in effect, the PL/I error condition may be raised. Output could be erroneous or some unpredictable error could result. In most cases, this problem can be circumvented in one or more of the following ways:

1. Restricting the region size to less than 256 megabytes.

   a. Restrict the PL/I program to less than 256 megabytes.

      - See MVS/XA System Programming Library: System Modifications
      - See MVS/XA System Programming Library: User Exits.

   b. With MVS/SP 2.1.2, use the REGION parameter by:

      - Finding the system "high water mark" above 16 megabytes.
      - Setting REGION = 256M - "high water mark".

2. In the case of variables which would be in a heap area if the HEAP option were to be used, either refraining from use of the HEAP option or using it with the BELOW sub-option.

3. In the case of STATIC variables, giving the load module the RMODE(24) attribute regardless of its AMODE attribute.

4. In cases where the above circumventions are infeasible, making sure that all bit variables have the AUTOMATIC attribute, since all AUTOMATIC storage is located below 16 megabytes.

5. Reserve space using a character field. Move the character field to an AUTOMATIC character field that can then be redefined as bit.

UNUSUAL ARRAY DECLARATIONS

Certain relatively unusual array declarations, in particular declarations in which both the lower and upper subscript bounds are negative numbers, may fail in 31-bit addressing mode on MVS/XA if the arrays happen to be allocated at very high address values in virtual storage.

PL/I addresses arrays by generating an address called the virtual origin. Regardless of addressing mode, the virtual origin is neither a 24-bit address nor a 31-bit address; it is a signed binary full-word. It is never used as a storage address; it is only a term in the computation of a storage address. The virtual origin is that address at which the array element whose

subscript values are all zero would be located, whether or not such an array element actually exists.

If the lower and upper subscripts of an array are both positive, then the virtual origin lies at a lower address than any actual array element. In fact the value of the virtual origin address may be negative, and PL/I object code and the PL/I library are designed to handle this eventuality.

If the lower bound is negative and the upper bound is positive, then the virtual origin is within the array, and thus addressable without difficulty.

If the lower bound and the upper bound are both negative numbers, then the virtual origin lies at an address higher than any actual element of the array. Such an address could potentially lie above the highest address defined by 31-bit addressing. (The 24-bit addressing mode counterpart, that the address lies above 2**24, is not a problem since the virtual origin address is a signed binary full-word.)

This problem is extremely unlikely to arise, since few arrays are declared with both lower and upper subscript bounds negative, and the array would have to be located at a very high virtual address.

If the problem arises and either the SPIE or STAE execution-time option is in effect, the PL/I error condition could be raised when object code or library code attempts to compute the address of the virtual origin. Output could be erroneous or some unpredictable error could result.

One circumvention would be to force the array to be located at a lower address in virtual storage. This could be done by one or more of the following techniques:

- Changing the order in which ALLOCATE statements for different variables in the program are issued.
- Refraining from the use of the HEAP option, or using the HEAP option with the BELOW sub-option.
- Giving the load module the RMODE(24) attribute.
- Giving the array in question the AUTOMATIC attribute.

If the above approaches are inadequate, undesirable, or infeasible, then it will be necessary to change the declaration of the array so that at least the upper bounds of the array have non-negative values and to adjust the subscript expressions accordingly.

INTERLANGUAGE COMMUNICATION

PL/I Release 5 supports interlanguage communication with both COBOL and FORTRAN. The language products supported include OS FORTRAN-(H) Extended, VS FORTRAN, OS/VS COBOL, and VS COBOL II. As has been the case in the past, interlanguage calls to COBOL and/or FORTRAN cannot be made from FETCHed procedures.

If all the object modules concerned were compiled by PL/I Release 5, VS FORTRAN, or VS COBOL II, then the resultant load module can have the AMODE(ANY) and RMODE(ANY) attributes.

PL/I STATIC EXTERNAL variables can be bound to the same storage locations as FORTRAN named COMMON whether the RMODE of the load module is RMODE(24) or RMODE(31), provided that the FORTRAN COMMON is not VS FORTRAN "dynamic COMMON".

LIMITS ON SIZES

Limits on subscript sizes and string sizes are unchanged in PL/I Release 5. Subscripts are still restricted to FIXED BINARY (15). However, the actual size of a multi-dimensional array can be much greater on PL/I Release 5 by making use of the storage available above 16 megabytes.

OBJECT CODE AND LIBRARY MODULES COMPATIBILITY

The requirements for mixtures of different levels of object code, resident library modules, and transient library modules have not been changed by Release 5. Neither object modules nor resident library modules can be later than the transient library, and all the transient library modules must be at the same level. No object module can be at a later level than the resident library, and all the resident library modules must be at the same level.

Thus if even one procedure in a load module has been compiled on Release 5, then the Release 5 Resident Library and Transient Library are required.

In addition, Release 5 adds the requirement that when Release 5 object modules are link edited with object modules produced by the PL/I Checkout Compiler R3, the load module must be link edited with the PL/I Resident Library R5; that is, PLICMIX cannot be used.

PL/I object modules compiled by prior releases of the PL/I Optimizing Compiler and/or the PL/I Checkout Compiler Release 3 can be link edited with PL/I Release 5 object modules and/or PL/I Release 5 Resident Library modules.

Since the pre-Release 5 object modules, however, contain instruction sequences which are incompatible with execution in 31-bit addressing mode, load modules containing pre-Release 5 object modules must be given the attributes AMODE(24) and RMODE(24).

OTHER CHARACTERISTICS OF RELEASE 5 IN MVS/XA

The general constraints imposed by MVS/XA and the specific characteristics of PL/I Release 5 define additional characteristics of PL/I Release 5 in the MVS/XA environment:

1. The PL/I Transient Library interfaces to a number of system service and data management facilities which must be invoked in 24-bit addressing mode. This is made possible by giving most of the modules in the Transient Library the attributes AMODE(ANY) and RMODE(24), thus meeting the first general MVS/XA constraint described under "Use of MVS/XA Facilities by PL/I Release 5" on page 457. The mode switches to and from 24-bit addressing mode are not directly visible to your program. The XA instructions that perform these mode-switches are bypassed, of course, on non-XA systems.

2. As is the case with prior releases of the PL/I Transient Library, the modules that comprise it are reentrant and can be made resident in your system. The RMODE(24) attribute will force them to reside in the link pack area below 16 megabytes.

3. The PL/I Shared Library is supported by PL/I Release 5 on both XA and non-XA systems; however the load modules which actually contain the shared Resident Library modules must reside below 16 megabytes on MVS/XA unless all users of the shared library have the AMODE(ANY) attribute.

   For most PL/I Release 5 users, at least initially, these load modules will require the attributes AMODE(ANY) and

RMODE(24), and will reside in the MVS pageable link pack area below 16 megabytes.

If all users of the shared library have the AMODE(ANY) attribute, then the shared library load modules can be given the RMODE(ANY) attribute and placed in the extended link pack area above 16 megabytes. This situation results from the fact that no mode-switching is done within the PL/I Resident Library.

TOTAL OPTION

The TOTAL option of RECORD I/O, which under earlier releases of the PL/I Optimizing Compiler caused in-line code to be generated for certain I/O statements for certain types of files and datasets, has been given a different implementation on PL/I Release 5.

The code formerly generated directly accessed data management control blocks and directly called OS data management. These functions must generally be performed in 24-bit addressing mode on MVS/XA.

The Release 5 implementation generates code to call special "fast path" modules in the PL/I Transient Library, so that mode-switching can be performed if necessary. This implementation results in a longer instruction path than on prior releases, but is still significantly faster than not using the TOTAL option.

LOCATE MODE I/O

The use of PL/I LOCATE mode I/O may require the use of extra buffers under MVS/XA. Extra buffers are required when the data is located above 16 megabytes, and data management requires that it be below 16 megabytes.

FETCH/RELEASE CONSIDERATIONS

PL/I Release 5 supports the PL/I FETCH/RELEASE facility. No special considerations apply to this support when both the fetching load module and the fetched load modules have the AMODE(ANY) attribute or both have the AMODE(24) attribute. However, PL/I Release 5 supports the fetching of a load module which has a different AMODE attribute than that of the fetching load module. PL/I will perform the mode-switches in this case, and the following constraints apply:

1. If any fetched module is to execute in 24-bit addressing mode, then the fetching module must be loaded into storage below 16 megabytes, and thus must have the RMODE(24) attribute regardless of its AMODE attribute.

2. It is your responsibility as the programmer to ensure that any variables passed as parameters to a fetched procedure are addressable in the AMODE of the fetched procedure. Thus for any fetched load module which is to be executed in 24-bit addressing mode:

   - If any parameter resides in a HEAP area because the HEAP option is in effect, then the BELOW sub-option of the HEAP option must be specified.

   - If any parameter resides in STATIC storage of the fetching load module, then the fetching load module must have the RMODE(24) attribute so that its STATIC storage will be below 16 megabytes.

   - No special considerations apply to parameters with the AUTOMATIC attribute, since AUTOMATIC storage for all procedures resides below 16 megabytes. If the first two

constraints cause problems, then one solution is to copy the variable to a like variable with the AUTOMATIC attribute and pass the copy to the fetched AMODE(24) procedure.

3. PL/I Release 5 object modules may be link edited into overlay load modules and executed as overlay load modules on either XA or non-XA systems, but such modules have the attributes AMODE(24) and RMODE(24).

When a PL/I program fetches another PL/I procedure, it is possible for a condition to arise in the fetched procedure for which a PL/I ON-unit was established in the fetching procedure.

PL/I Release 5 imposes the restriction that if an ON-unit is established while the current addressing mode is 24-bit, and the condition is raised while the addressing mode is 31-bit, then the ON-unit will not be entered. This is because PL/I must invoke the ON-unit in the addressing mode in which it was established.

If the ON-unit was established in 24-bit addressing mode but the condition arose in 31-bit addressing mode, the code and data required to process the error may not even be addressable in 24-bit addressing mode.

THE PL/I NULL POINTER AND MVS/XA

PL/I provides two data types that point to other user variables: pointers and offsets. Pointer variables and offset variables are similar in function but not identical.

Pointers point to data items in storage for a particular execution of a PL/I program. A pointer value may be set to point to an already existing variable by use of the ADDR built-in function, set via READ or LOCATE statements with the SET option, or set via an ALLOCATE statement which allocates a BASED variable.

Offset variables point to, and only to, PL/I variables which are allocated within PL/I AREA variables by use of the ALLOCATE statement with the IN (area-name) option. If the entire area variable is written via a PL/I WRITE statement, then the record thus written may be read and processed by either the same or a different execution of the program, or read and processed by some other program. The value of an offset variable is meaningful with respect to the beginning of the area, not storage in general.

For a more detailed discussion of pointers, offsets, and based storage in general, see the PL/I Language Reference Manual.

While it is clear that the value of a pointer (or offset) that points to a variable is the storage address of the variable (or the relative address within a PL/I area of the variable), there is a need to have a way to indicate that a pointer or offset does not point to anything at all.

This is accomplished by assigning to a pointer or offset variable the value provided by the PL/I NULL built-in function. This value is called the "null pointer" or "null offset".

While a non-null value of a pointer or offset variable points to a storage location, the null pointer is merely a token which means, "This pointer (or offset variable) does not point to anything".

The valid uses of NULL are:

1. To assign NULL to a pointer or offset variable which does not point to a storage location

2. To compare the current value of a pointer or offset variable to NULL to determine whether or not the pointer or offset variable currently points to a storage location.

It is a programming error to use the null pointer as though it were the address of something.

As a matter of implementation, programs compiled by the PL/I Optimizing Compiler have always used the hexadecimal value "FF000000" for the null pointer and the null offset variable. This is the value returned by the NULL built-in function. This implementation is retained in PL/I Release 5.

The convention defined by MVS/XA for use of the high-order bit to indicate addressing mode applies to branch addresses which are to be used in 370/XA mode-switching branch instructions, not to data addresses. PL/I pointers point to data; they do not represent branch addresses.

Any non-null pointer or offset value generated by PL/I has the high-order bit off (zero). Therefore, no pointer or offset value generated by PL/I can ever be confused with the null pointer.

If you provide to PL/I a pointer value which might have the high-order bit on and is intended to be used in 31-bit addressing mode, you should ensure that the high-order bit is turned off, at least before you compare it to the null pointer.

Note that there is no reason to compare a pointer to NULL if you know the pointer or offset variable points to something in storage.

As a matter of good programming practice, if you pass a based variable in a subroutine call, and it is possible that the based variable may not exist, you should pass a pointer to the based variable, not the based variable itself. Then the parameter list constructed by PL/I will contain the address of the pointer, and the called program can compare the pointer to the NULL built-in function to see if the based storage exists.

If you pass the based variable itself, and the based variable does not exist, then any value in the parameter list for the non-existent based variable is garbage, and any reference to the non-existent based variable is in error.

Such a program is inherently invalid in either 24-bit addressing mode or 31-bit addressing mode, and would be invalid no matter what value is used for the null pointer.

PL/I pointers or offset values that do not point to an actual variable in storage are not initialized to NULL by PL/I. Any such initialization must be done by you. Initializing such pointers or offset variables to NULL is good programming practice.

APPENDIX G. IMS CONSIDERATIONS FOR PL/I RELEASE 5

Many IBM customers use PL/I as an application programming language in the IMS environment. The interface between PL/I and IMS has traditionally resided entirely within IMS, and the documentation of how a programmer would write either a batch IMS program or an online IMS/DC transaction in PL/I has been provided entirely by IMS documentation. PL/I releases prior to PL/I Release 5 provided neither special support nor special documentation for the IMS user.

PL/I Release 5 provides some special support in the IMS/VS Release 1.2 and IMS/VS Release 1.3 environments, including enhanced PL/I - IMS error handling support for both these releases and support (with IMS Release 1.3 only) for the 31-bit addressing capabilities of MVS/XA.

BACKGROUND FOR ENHANCED PL/I-IMS ERROR HANDLING

The IMS environment, especially the IMS/DC environment, is very sensitive to errors and error handling issues, since a failing IMS transaction or program can potentially contaminate an IMS database. For this reason, it is essential that IMS know about the failure of a transaction or program that has been updating a database so that it can back out any updates made by that failing program.

PL/I provides extensive error handling facilities to the application programmer, but in the absence of coordination of error handling facilities between IMS and PL/I, the IMS implementers have found it necessary to recommend to PL/I programmers writing IMS programs that they disable much of the PL/I error handling function available in the PL/I language.

This recommendation has taken the form of instructing the PL/I programmer to:

- Execute PL/I programs with the NOSPIE and NOSTAE execution time options in effect rather the SPIE and STAE options. (This means that it has not been possible (or at least, supported) to get control in user-coded PL/I ON-units after any error other than a PL/I software-detected condition.)

- Provide an installation-modified version of a PL/I module called IBMBEERA, described in the PL/I Installation Manual, so as to cause any PL/I program terminated in error to be terminated via an operating system ABEND request. (Such a termination in error could only arise from a PL/I software-detected condition anyway.)

These injunctions sought to prevent these problems:

1. If a PL/I program was executed with the STAE option, then PL/I would have issued an operating system STAE request to try to get control after an abend occurred. What would happen then depended on the release of IMS and the version of the operating system in use, as follows:

   - If the release of IMS and the version of the operating system were such that IMS was also using an operating system STAE request to get control after an error, then the PL/I STAE request and the IMS STAE request would interfere with each other.

     IMS would in this case re-issue its own STAE macro each time the PL/I program called IMS. This required in turn that IMS re-instate the PL/I STAE request before returning to PL/I.

This represents enormous execution-time overhead, but it ensured that if the abend arose when IMS was in control, IMS could get control to terminate the transaction and back out any updates that the failing program had made to IMS databases.

If the PL/I program itself was in control when the abend occurred, then the PL/I error handler (and thereafter some PL/I ON-unit if one had been established) would get control.

If the user at that point repaired the error or issued a roll-back call to IMS to back out updates by the transaction, then no harm was done.

If the user did not repair the error, however, but took normal return from an ERROR ON-unit (or executed the program with the STAE option and did not even establish an ERROR ON-unit), then the program would appear to IMS to have terminated normally when in fact it had failed and might have contaminated a database.

- If the operating system was MVS/370 or MVS/XA, and the IMS release was a fairly recent one, then IMS had established its own error handling environment not by use of a STAE request but by use of the newer MVS ESTAE request. PL/I, of course, had issued its usual STAE request.

  If both STAE and ESTAE requests are in effect simultaneously, then the ESTAE requestor gets control, not the STAE requestor, when an abend occurs. In this situation, therefore, even if the abend arose while the PL/I program was in control, the PL/I error handler would never get control. Thus no code that you put into your PL/I error ON-unit could ever get control either.

  Since a program written to be executed with the STAE option could very well contain code intended to be executed in case of error, and that code could be important to the integrity of the overall application, the fact that this code would no longer be executed represents a profound and potentially dangerous change in the "semantics" of your program and the IMS application of which it is a part. Moreover, this change in semantics could be well hidden and quite unexpected, occurring after a change of IMS releases or a change from VS1 to MVS.

To resolve these problems, the advice was given to execute the PL/I program with the NOSTAE option.

2. If the PL/I program was being executed with the PL/I SPIE option in effect, then it was intended by IMS that the IMS region controller be told that SPIE was in effect so that IMS could alternately reinstate its own SPIE request and PL/I's SPIE request.

   As was the case with STAE above, this represented enormous overhead. Furthermore, it opened up some of the same integrity exposures described above for STAE.

   To resolve these problems, the advice was given to execute the PL/I program with the NOSPIE option.

3. In any case a PL/I software-detected error (e.g., CONVERSION) could arise in the program and represent an error. This condition, if not corrected, could cause ERROR to be raised and could then cause the program to terminated in error.

   IMS would not ordinarily know that the program had terminated in error, and thus could not back out updates made by the failing program.

To resolve this problem the advice was given to provide an installation-modified version of IBMBEERA to force any non-normal termination of a PL/I program to result in a system ABEND request.

PL/I Release 5 provides support to give back to the PL/I programmer the error handling facilities of PL/I in those cases in which the ABEND or program check occurs within the PL/I program as opposed to IMS. Specifically, on PL/I Release 5 with IMS Release 1.2 or 1.3:

- You may execute your PL/I program with the STAE and/or SPIE options, provided that it interfaces to IMS by calling PLITDLI or ASMTDLI or EXECDLI (as opposed to calling some private IMS interface) and provided that you recompile every PL/I program in the transaction load module using PL/I Release 5. This implies, of course, that you must then re-link-edit the load module with the Release 5 PL/I Resident Library and execute it with the Release 5 PL/I Transient Library.

- If you use the SPIE option, you need not tell the IMS region controller that your program is issuing SPIE macros.

- PL/I will route calls to (and returns from) PLITDLI and ASMTDLI through a PL/I library routine and keep track of transfers of control between your PL/I program and IMS. Thus if an abend or program check occurs and the PL/I error handler gets control, it can tell if the problem arose on the IMS side of the interface or on the PL/I side of it.

- If a program check or abend occurs in IMS, then when the PL/I exception handler gets control it will immediately "percolate" the error back to IMS. No PL/I condition will be raised, no PL/I ON-unit will get control, no PL/I message will be produced, and IMBEERA will not get control.

- If a program check occurs in the PL/I program rather than in IMS, then all the facilities of PL/I error handling apply, provided that you meet the certain conditions when you code your program. For any error condition that arises, you must do one of the following:

  1. Resolve the error completely so that the application can continue, or

  2. You must tell IMS to back out the program's updates by issuing a rollback call to IMS and then terminate the program, or

  3. You must make sure that the program terminates in error and that an installation-modified IBMBEERA applies which will cause any non-normal PL/I program termination to result in an operating system ABEND request.

  The kinds of errors you are most likely to be able to fix in your program are PL/I software-detected conditions such as CONVERSION, program check interruptions which raise the PL/I OVERFLOW, UNDERFLOW, FIXEDOVERFLOW, or ZERODIVIDE conditions, and a program check interruption for a data exception (which raises ERROR with ONCODE 8097). It is relatively unlikely that you can resolve other types of program checks or system abends in your program.

Any IMS program which invokes IMS via some private interface or which you do not choose to recompile and re-link-edit on PL/I Release 5 should be executed with NOSPIE and NOSTAE in effect. Even so, it should either contain code to issue a rollback call to IMS before terminating after an error, or it should be executed with an installation-modified IBMBEERA which ensures that any non-normal PL/I program termination results in an operating system ABEND request.

PL/I RELEASE 5, IMS 1.3, AND MVS/XA

When PL/I programs compiled by PL/I Release 5 are executed with IMS 1.3 or subsequent, the PL/I programs can be executed in either 24-bit or 31-bit addressing mode. Such programs can have load module attributes of RMODE(24) and AMODE(24), of course, but they can also have the attribute of AMODE(ANY). With AMODE(ANY), the RMODE attribute can be either RMODE(24) or RMODE(ANY).

IMS imposes the restriction that all parameters passed to IMS in a call to PLITDLI or ASMTDLI except the parameter count must be located below 16 megabytes. All storage areas that are referred to in the paramater list of a COBOL, PL/I, or Assembler application program call to IMS/VS Version 2, Release 1, or IMS/VS Version 1, Release 3, must reside in virtual storage below 16 megabytes. This includes the function, the I/O area, the SSA(s), the MOD name, and the destination name. The parameter count field, if present, may optionally reside in the extended virtual storage area. Note that the names PLITDLI and ASMTDLI are interpreted to mean IMS interfaces; if they are being used in any other way in a program, they must be changed. The PL/I program can meet this condition by using the following techniques in any combination for the parameters passed to IMS:

- Placing IMS parameters in AUTOMATIC storage. All AUTOMATIC storage is below 16 megabytes on MVS/XA, regardless of the RMODE or AMODE of the program.

- Placing IMS parameters in CONTROLLED storage or in BASED storage which is allocated by PL/I ALLOCATE statements, provided that such storage is held below 16 megabytes.

  Such variables can be forced to reside below the line by using HEAP(0) to force them to be allocated in the ISA or an extension to the ISA, or by using a non-zero value for the HEAP size and supplying the BELOW sub-option of HEAP to cause the heap area and any extensions to it to be placed below the line. (Certain IMS variables addressed by the PL/I program as BASED variables were actually allocated by IMS and passed to PL/I by IMS in the first place. These variables are placed below the line by IMS.)

- Placing IMS parameters in STATIC storage and using a load module attribute of RMODE(24) to force the load module (and thus STATIC storage) to be placed below the 16 megabytes.

  While the use of RMODE(24) and IMS parameters in STATIC storage can meet the IMS requirement for IMS parameters to be below the line, this technique defeats one of the most attractive possibilities for the IMS user to employ PL/I Release 5 in 31-bit addressing mode on MVS/XA.

For many IMS users, the storage required at execution time for any particular IMS transaction may be fairly small, and the most attractive way to use PL/I Release 5 is actually to code the PL/I application programs to be reentrant (i.e., to code them so that they do not alter any STATIC variable and to specify "OPTIONS(REENTRANT)" on the PROCEDURE statement).

If for the special case of IMS 1.3 and subsequent, the additional constraint is imposed that STATIC variables not be passed to IMS via calls to PLITDLI or ASMTDLI, then the PL/I programs can be given the load module attributes AMODE(ANY) and RMODE(ANY) and perhaps placed in the Extended Link Pack Area of MVS/XA.

This can eliminate program loading time, speed up IMS initialization and restart, and provide the additional integrity that results from having application programs reside in protected storage.

Special Characters

A

B

C

D

E

M

N

O

P

Q

R

S

T

U

V

W

X

Numerics

Reader's Comment Form

OS PL/I Optimizing Compiler:
Programmer's Guide
SC33-0006-7

This manual is part of a library that serves as a reference source for systems analysts, programmers, and operators of IBM systems. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate.

Note: *Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.*

Please use pressure sensitive or other gummed tape to seal this form.

List TNLs here:

If you have applied any technical newsletters (TNLs) to this book, please list them here:

Last TNL _____

Previous TNL _____

Previous TNL _____

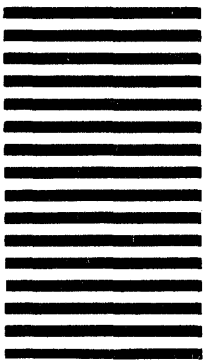
Fold on two lines, tape, and mail. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail directly to the address in the Edition Notice on the back of the title page.) Thank you for your cooperation.

SC33-0006-7

Reader's Comment Form

Fold and tape | Please do not staple | Fold and tape

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



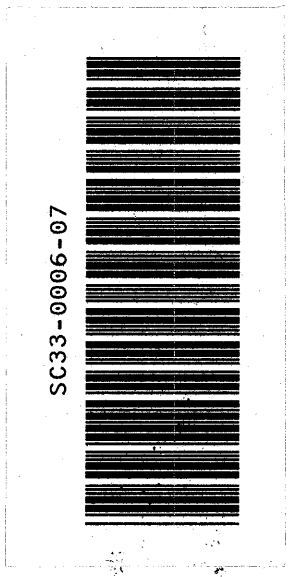
BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
P.O. Box 50020
Programming Publishing
San Jose, California 95150

Fold and tape | Please do not staple | Fold and tape

OS PL/I Optimizing Compiler: Programmer's Guide (File No. S370-29) Printed in U.S.A. SC33-0006-7



SC33-0006-7

OS PL/I Optimizing Compiler: Programmer's Guide (File No. S370-29) Printed in U.S.A. SC33-0006-7

IBM

SC33-0006-07