SC33-0008-6
File No. S360/S370-29

Program Product

DOS PL/I Optimizing Compiler: Programmer's Guide

Optimizing Compiler	**5736-PL1**
Resident Library	**5736-LM4**
Transient Library	**5736-LM5**

(These program products are also available as composite package 5736-PL3)

Release 6.0

Seventh Edition (May 1984)

This is a major revision of, and makes obsolete, SC33-0008-5.

This edition applies to Release 6.0 of DOS PL/I Optimizing Compiler, Program Product 5736-PL1, 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.

PREFACE

This publication is a guide to the use of the PL/I Optimizing Compiler (Program No. 5736-PL1) in a batch environment under your operating system. It explains how to use the compiler to compile and execute PL/I programs, and describes the features of the operating system that may be required by a PL/I programmer. It does not describe the language implemented by the compiler or explain how to use the compiler with the Conversational Monitor System (CMS); these are functions of the manuals listed under "Associated Publications" below.

During execution of a PL/I program, the optimizing compiler employs subroutines from the DOS PL/I Resident Library (Program No. 5736-LM4) and the DOS PL/I Transient Library (Program No. 5736-LM5). This programmer's guide assumes the availability of these program products.

Different release levels of the DOS PL/I Optimizing Compiler and the PL/I Resident and Transient Libraries will provide compatible execution provided that the following conditions are satisfied:

- The release and service level of the transient library are equal to or greater than the release and service level of the resident library.
- The release and service level of the resident library are equal to or greater than the release and service level of the compiler.

Chapter 1, "Running a PL/I Program," through Chapter 3, "How to Define a Data Set," of this programmer's guide cover basic topics, and are intended primarily for casual (nonspecialist) programmers or for newcomers to IBM systems. The reader is assumed to have only an elementary grasp of PL/I and the basic concepts of data processing. These chapters introduce the IBM Disk Operating System and Disk Operating System with Virtual Storage (DOS/VS), and explain how to run a simple PL/I program and how to define a simple data set.

The rest of the manual contains more detailed information on the optimizing compiler, and provides general guidance and reference information on operating system features that are likely to be required by the PL/I applications programmer.

Chapter 4, "The Optimizing Compiler" on page 14, describes the optimizing compiler, the data sets it requires, its optional facilities, and the listings it produces. Chapter 5, "The Linkage Editor" on page 43, contains similar information about the linkage editor, which is always needed to prepare a PL/I program for execution.

Chapter 6, "Program Library Creation and Maintenance" on page 66, is concerned with the various types of program library available under the Disk Operating System.

Chapter 7, "Data Sets and Files," through Chapter 10, "Using VSAM Data Sets from PL/I," are concerned with the various types of data sets that can be created and accessed by a PL/I program, and explain how to define these data sets.

Chapter 11, "Program Checkout," describes the facilities available for debugging PL/I programs.

Chapter 12, "Linking PL/I and Assembler Language Modules," explains how to write programs that contain a combination of PL/I and assembler-language modules.

Chapter 13, "Checkpoint/Restart," and Chapter 14, "PL/I SORT," are concerned with the use of built-in subroutines included in the optimizing compiler to provide a direct interface between PL/I programs and the operating system checkpoint/restart and sort facilities.

Chapter 15, "Communication with COBOL, FORTRAN, and RPG," describes how the PL/I interlanguage facilities permit communication, at execution time, between programs compiled by FORTRAN, COBOL, and RPG compilers, and executed using the corresponding libraries.

Chapter 16, "Using PL/I on CICS," describes the use of PL/I in conjunction with CICS facilities.

Appendixes supply reference information.

ASSOCIATED PUBLICATIONS

The language implemented by the DOS PL/I Optimizing Compiler is described in the following publication:

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

For information on how to use the compiler under CMS, refer to:

- DOS PL/I Optimizing Compiler: CMS User's Guide, SC33-0051

The PL/I Optimizing Compiler, its facilities, and its requirements are described in the following DOS publication (which also contains a comparison of the language implemented by this compiler).

- DOS PL/I Optimizing Compiler: General Information, GC33-0004

Compile-time and execution-time messages for this compiler are documented in the following DOS publications:

- DOS PL/I Optimizing Compiler Messages, SC33-0021
- DOS PL/I Transient Library: Messages, SC33-0005

Additional information about the object programs generated by the DOS PL/I Optimizing Compiler and the PL/I resident and transient modules is contained in the following DOS publication:

- DOS PL/I Optimizing Compiler: Execution Logic, SC33-0019

Information about installing and operating the DOS PL/I Optimizing Compiler, including both system generation and storage requirements, is contained in the following DOS publication:

- DOS PL/I Optimizing Compiler: Installation Guide, SC33-0020

The following manuals describe the control statements that relate to the SCP (system control programming) and the VSE/Advanced Functions of DOS/VSE (Disk Operating System/Virtual Storage Extended).

- DOS/VSE System Control Statements, GC33-5376
- VSE/Advanced Functions Control Statements, SC33-6095

Information on DOS/VSE I/O macros can be found in:

- VSE/Advanced Functions Macro Reference, SC24-5211

The following publication provides all the VSAM information needed to use Access Method Services in order to establish and maintain VSAM data sets.

- DOS/VS Access Method Services User's Guide, GC33-5382

The types of labels that may be written on magnetic tape or disk by DOS/VSE are defined and described in the following manuals:

- DOS/VSE DASD Labels, GC33-5375
- DOS/VSE Tape Labels, GC33-5374

The following publications contain information for assembler, COBOL, and PL/I application programmers for preparing programs using CICS/VS commands to execute under either CICS/DOS/VS or CICS/OS/VS.

- Customer Information Control System/Virtual Storage (CICS/VS) Version 1, Release 3: Application Programmer's Reference (Command Level), SC33-0077
- Customer Information Control System/Virtual Storage (CICS/VS) Version 1, Release 3: Application Programmer's Reference (Macro Level), SC33-0079

For information about the capabilities of the IBM 3800 Printing Subsystem, refer to:

- DOS/VS IBM 3800 Printing Subsystem, GC26-3900

For detailed information on the functions and capabilities of the IBM 3881 Optical Mark Reader, refer to:

- IBM 3881 Optical Mark Reader Models 1 and 2 Reference Manual and Operator's Guide, GA21-9143

For definitions of terms used in this manual, see the following publication:

- IBM Data Processing Glossary, GC20-1699

SYNTAX NOTATION

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

INDUSTRY STANDARDS

The DOS PL/I Optimizing Compiler is designed according to the specifications of the following industry standards as understood and interpreted by IBM as of October, 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

MAY 1984

NEW PROGRAMMING FEATURE

Support for the IBM 3370 Model 2 and 3380 Direct Access Storage devices is provided under DOS/VSE with VSE/Advanced Functions, Release 3.5.

SERVICE CHANGE

The section, "Restrictions on Input/Output by FORTRAN Routines" on page 261, has been added to Chapter 15, "Communication with COBOL, FORTRAN, and RPG."

Other miscellaneous corrections and clarifications have been made throughout the manual.

SEPTEMBER 1981

NEW PROGRAMMING SUPPORT

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

NEW PROGRAMMING FEATURE

Support for the IBM 3375 Direct Access Storage device is provided under DOS/VSE with VSE/Advanced Functions, Release 3.

SERVICE CHANGES

This edition is for use with the new _OS and DOS PL/I Language Reference Manual_. Information moved from the _DOS PL/I Optimizing Compiler Language Reference Manual_ into this edition of the programmer's guide includes:

- The ENVIRONMENT attribute, data transmission statements, and related topics.
- "Associating Data Sets with Files" on page 91.
- Chapter 15, "Communication with COBOL, FORTRAN, and RPG" on page 247.

The following updates have been made throughout the manual:

- The SIZE operand is now required on the EXEC system control statement.
- The LIBDEF control statement may be used in place of SYSSLB, SYSRLB, and SYSCLB when operating under DOS/VSE with VSE/Advanced Functions.
- The LBLTYP system control statement is no longer required when operating under DOS/VSE with VSE/Advanced Functions.

In addition, Appendixes B, D, G, H, and I have been removed. A new Appendix B, "VSAM Background," and a new Appendix D, "CICS System Information," have been added.

- Information previously found in Appendix B can be found in _DOS/VSE System Control Statements_.
- Information previously found in Appendix D can be found in _DOS/VSE DASD Labels_ and _DOS/VSE Tape Labels_.
- Information previously found in Appendixes G and H have been incorporated into the body of the manual.

Other miscellaneous corrections have been made throughout the publication.

DECEMBER 1979

VSE/VSAM SPACE MANAGEMENT FOR SAM FEATURE

DOS/VSE with VSE/Advanced Functions, Release 2, supports the VSE/VSAM Space Management for SAM Feature of VSE/VSAM Release 2. Information on using this feature with DOS PL/I is included in this manual.

Other changes and corrections have also been made throughout the manual.

JANUARY 1979

DEVICE SUPPORT: FIXED BLOCK DEVICES

Support for fixed block devices is provided under DOS/VSE with VSE/Advanced Function, Release 1.

Other changes and corrections have also been made throughout the manual.

AUGUST 1977

DEVICE SUPPORT: 3350 AND 3330-11

NEW PROGRAMMING FEATURE

DOS/VS Release 34 provides support for the 3330-11 and 3350 direct access storage devices.

CONTENTS

FIGURES

CHAPTER 1. RUNNING A PL/I PROGRAM

In describing how to run a PL/I program, this chapter mentions some of the features of the IBM Disk Operating System (DOS) and additional features available when running under a Virtual Storage operating system (DOS/VS). For readers unfamiliar with the Disk Operating System, these features are described in Chapter 2.

This chapter describes how to run a program that uses card input and printed output. Chapter 3 describes how to define simple consecutive data sets for creation and access by a PL/I program.

Using the DOS PL/I Optimizing Compiler to run a PL/I program involves three steps:

1. Compilation

2. Link-editing

3. Execution

The first step, compilation, is necessary to create an object module from the PL/I source program. The second step, link-editing, is necessary to combine the PL/I object module with object modules from the PL/I library or other programs to form an executable program. The third step is simply the execution of the executable program created in the link-editing step.

When you submit a PL/I program as a job for execution, you must supply the appropriate DOS job control statements so that the operating system can initiate the required functions in the correct order. The basic job control statements are:

- The JOB statement—initiates the job.
- The OPTION statement—specifies options for the job.
- The EXEC statement—initiates loading and execution of a program.
- The end-of-data statement (/*).
- The end-of-job statement (/&).

A typical sequence of DOS job control statements, PL/I source statements, and data for the execution step follows:

```
// JOB PLITEST
// OPTION LINK
// EXEC PLIOPT,SIZE=64K
       .
       .
       .
     PL/I source statements
       .
       .
/*
// EXEC LNKEDT
// EXEC ,SIZE=64K
       .
       .
     data
       .
       .
       .
/*
/&
```

If you use punched cards as input to your system, all job control statements start in column 1. Apart from the end-of-data (/*) and the end-of-job (/&) statements, all job control statements have // in the first two columns, followed by at least one blank, and a job control keyword such as EXEC, OPTION, or JOB.

The JOB statement is always the first statement in a DOS job. Code it as follows:

```
// JOB jobname [job-identification]
```

where jobname is any name of up to eight alphameric characters, the first of which must be an alphabetic character, and job-identification is an optional field that can contain any additional comment or installation-defined information about the job that you may want to record.

The OPTION statement with the LINK option must be included to signify that the compiler should prepare an object module for link-editing. Code the OPTION statement as follows:

```
// OPTION LINK
```

The EXEC statements must be present to invoke the optimizing compiler for the compilation step, the linkage editor for the link-editing step, and the executable PL/I program for the final step. To invoke the compiler, code the EXEC with the SIZE option as follows:

```
// EXEC PLIOPT,SIZE=nK
```

where n specifies the amount of storage that is to be used by the program.

The DOS PL/I Optimizing Compiler can also be invoked from a cataloged procedure, using the form PROC=procedure-name.

To invoke the linkage editor, code the EXEC statement as follows:

```
// EXEC LNKEDT
```

To invoke the executable PL/I program, code the EXEC statement as follows:

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

The last use of the EXEC statement need not include the name of a program as an operand since an EXEC statement without a program name operand will cause the last executable program that was link-edited to be executed. Hence, this form of the EXEC statement will normally be used only immediately following a link-editing step. However, the SIZE operand must be used in either case:

```
// EXEC program-name,SIZE=nK

        or

// EXEC ,SIZE=nK
```

The end-of-data statement (/*) must be used to indicate the end of the PL/I source program and the end of any data that is used by the PL/I program when it is executed.

The end-of-job (/&) statement must be used to indicate the end of the job.

A complete set of job control statements for a PL/I compilation, link-editing, and execution is given in the programming example in Figure 1 on page 3.

```
// JOB FIG0101
// OPTION LINK
// EXEC PLIOPT,SIZE=64K
 /* PROGRAMMING EXAMPLE TO PRINT THE SUM OF */
 /* PAIRS OF VALUES OBTAINED FROM PUNCHED   */
 /* CARD INPUT                              */

 TEST: PROCEDURE OPTIONS(MAIN);
       ON ENDFILE(SYSIN) GOTO ENDTST;
       DECLARE (A,B,C) FIXED DECIMAL(3);
 NEXT: GET FILE(SYSIN) DATA (A,B);
                    C=A+B;
                    PUT FILE(SYSPRINT) SKIP DATA(A,B,C);
                    GOTO NEXT;
 ENDTST: END TEST;
/*
// EXEC LNKEDT
// EXEC ,SIZE=64K
A=131  B=75;
A=2    B=907;
A=-14  B=14;
A=341  B=429;
A=-245 B=102;
A=999  B=-100;
/*
/&
```

Figure 1. PL/I Programming Example

When using the DOS PL/I Optimizing Compiler with DOS/VS, DOS/VSE, and DOS/VSE with Advanced Functions, the following are available:

Virtual storage
Multiple user partitions
Relocating loader
Catalogued procedures
Virtual Storage Access Method

With virtual storage, the address space available for problem programs is not limited by the physical size of main storage. Large programs no longer need to be structured by overlay techniques to fit into the available main storage, and multiprogramming operations are less constrained by the size of main storage.

Under DOS/VS, up to five partitions can be specified at system generation time. These are: BG, F4, F3, F2, and F1, in increasing order of priority. More partitions may be specified under DOS/VSE with Advanced Functions. The number of partitions is dependent upon the release level of your system.

The relocating loader can load a program at any address in any partition without the need to write self-relocating programs or to link-edit again. The DOS PL/I Optimizing Compiler will execute in any of the five partitions without the need for multiple copies of the compiler; output from the compiler will also execute in any partition.

DOS/VS makes available an additional library, the procedure library. This is used to contain cataloged procedures; for example, the job control statements necessary to compile, link-edit, and execute a program.

A description of PL/I support for VSAM data sets is given in Chapter 10.

PERFORMANCE

The methods of obtaining optimum performance depend on the system configuration and the workload of the machine, but some general guidelines are set out below.

COMPILE-TIME PERFORMANCE

The DOS PL/I Optimizing Compiler has been designed to adjust its processing to make full use of the storage available to it, whether real or virtual. The compiler uses its spill file when the storage available to it is insufficient; the point at which no spilling occurs depends on the size and content of the program being compiled.

The compile time in virtual mode is affected by the relative use of the compiler spill file and the page data set and the relative speed of the spill file device and the page data set device.

If the page data set is on a faster device than the compiler spill file, it may be better to use a large virtual partition and specify a SIZE option just large enough to avoid using the spill file. This situation may, however, be altered if other partitions are in contention for these devices. If the compiler spill file and the page data set are on the same drive, there will be contention between the two.

It is possible to compile some programs with less real storage than the 50K bytes design point of the compiler, but performance will be degraded. To minimize the overcommitment, the amount of virtual storage used should be restricted by specifying a SIZE option that is the larger of 50K bytes or the amount of real storage expected to be available.

EXECUTION-TIME PERFORMANCE

Some PL/I features that may not be practical under DOS are more feasible under DOS/VS because the constraint of real storage size is reduced; these include the STMT, FLOW, and COUNT compiler options, and the PUT DATA and CHECK statements. Use of these features will increase program size and may degrade performance, but could give savings in terms of reduced debugging time.

The performance of a program under DOS/VS and DOS/VSE depends on the address reference pattern; if this is localized, there will be less likelihood of system paging than if the whole virtual address space is referenced randomly. Use of PL/I block structuring helps to keep address references to object code localized.

Many of the hints in the _OS and DOS PL/I Language Reference Manual_ in the section on "Efficient Programming" are applicable to program performance under DOS/VS. The following points should also be considered when using PL/I for execution under a virtual storage operating system:

- Bit string operations and conversions done by library calls should be avoided.

- The inline I/O facility should be used wherever possible, and blocking factors should be increased as far as consistent with other requirements of the program.

- Large multidimensional arrays should be processed, or cross-sections passed as arguments, in row major order, not column major order.

- List-processing applications, in which data structures are chained together, may cause paging when the chain is followed. The ALLOCATION built-in function is implemented as a scan through a chain of allocations of controlled data.

- Near simultaneous references to variables declared in different blocks may cause paging.

- Care must be taken in creating load modules for programs as the layout of a load module becomes important when it is executed in virtual mode. Infrequently used modules such as IBMBERR, IBMBOCL, and IBMBPGR should be grouped away from more frequently used modules.

CHAPTER 2. INTRODUCTION TO DOS AND THE DOS PL/I OPTIMIZING COMPILER

The IBM Disk Operating System (DOS) consists of a control program and a number of service programs that together assist both the operator and the programmer in the use of IBM computers. An operating system relieves the programmer and operator of many time-consuming tasks, including the allocation of internal and external storage space, the control of input/output devices, and the control of the jobs being processed by the system.

It is often necessary to execute two or more programs as a related group, passing information created by one program as input to the following program. Such a group of programs can be executed as a self-contained job by using the batched job processing facilities of the Disk Operating System. Each program executed within the job is processed as a separate job step.

The control program supervises the execution of all other programs, and services the common requirements of these programs during execution. It has three main components:

1. Supervisor: The supervisor controls and coordinates all activity within the computing system.

2. Job Control: The job control program processes all job control statements and commands that enter the system.

3. Initial Program Load (IPL): The IPL routine is used by the operator to reactivate the operating system at the beginning of the day's processing, or after some other operating system or stand-alone program has been used.

The operating system allows many programs to occupy main storage simultaneously. Each program is loaded into a predefined area of main storage known as a _partition_. Under DOS/VSE with Advanced Functions, the number of partitions provided is dependent upon the release level of your system. One partition is known as the _background_ partition; the remaining partitions are the _foreground-1_ partition through the _foreground-n_ partition, respectively. A program can be executed in any partition as a job step within a job. Alternatively, a program can be initiated in any foreground partition by a command from the operator's console.

Programs executed concurrently in the various partitions compete with each other for the resources of the system, such as the central processing unit or an input/output channel. The allocation of such resources is controlled by the supervisor program. The relative priority of the program is set by the partition in which it is executed. The foreground-1 partition has the highest priority and the background partition has the lowest priority.

The service programs of the Disk Operating System include librarian and a number of utility programs; all these are described in separate publications which are listed in the preface.

The operating system provides facilities for the creation and maintenance of three different types of program libraries (collections of programs stored on a direct-access device). The libraries are the _core-image_ library, the _relocatable library_, and the _source statement library_.

A core-image library is used to hold executable (link-edited) programs; all programs to be executed under the control of the DOS supervisor program must be stored in a core-image library.

A relocatable library is used to hold relocatable (compiled or assembled, but not link-edited) object modules; object modules in a relocatable library can be readily incorporated into an executable program by the linkage editor.

A source statement library is used to hold sequences of source program statements that can be included in a compilation by the compiler (by means of the %INCLUDE statement). Such sequences of statements are termed books.

Each system will include a system core-image library, a system relocatable library, and a system source statement library. The system libraries will contain the system control programs supplied by IBM as components of the Disk Operating System. System libraries can also contain IBM program products and user-written programs. Alternatively, such programs can be held in additional private libraries. Private libraries can also be created for the three types of program libraries. Chapter 6 on page 66, gives further information about these libraries.

PL/I OPTIMIZING COMPILER

The DOS PL/I Optimizing Compiler is an IBM program product designed for use with the IBM Disk Operating System. This compiler translates PL/I source statements into machine instructions. The set of instructions resulting from the compilation of a PL/I external procedure is termed a relocatable object module. Before such a module can be executed, it must be processed by the linkage editor and placed in a core-image library.

The compiler can be used in any partition, but can only be initiated as a job step within a job.

LINK-EDITING PL/I PROGRAMS

The linkage editor is a service program that converts relocatable object modules into core-image modules (termed executable programs, or executable program phases in the case of multiphase programs). The linkage editor can obtain its input both from SYSLNK input stream and from a relocatable library.

A feature of the linkage editor is its ability to combine two or more relocatable object modules into a single executable program. This feature is particularly significant for a PL/I program because the optimizing compiler does not generate directly all the machine instructions that are required to present a source program. Instead, for frequently-used functions such as input/output, the compiler generates references to standard PL/I library subroutines. Many of these subroutines are in the DOS PL/I Resident Library; the remainder are in the DOS PL/I Transient Library. The linkage editor retrieves those resident library modules that are required for a PL/I program from the relocatable library and incorporates them into the executable program.

The subroutines in the transient library are not incorporated by the linkage editor into the executable program. These routines are loaded directly into storage from the transient library, which is held in the core-image library, only when required during the execution of the PL/I program. Many of the subroutines in the transient library are concerned with input/output and with error-handling.

CONVERSATIONAL MONITOR SYSTEM

The Conversational Monitor System (CMS) component of the Virtual Machine Facility/370 (VM/370) provides program creation, compilation, testing, and execution services by means of conversational time-sharing from remote terminals. Using the compiler under CMS permits many users of the same computer to develop programs concurrently. Instructions on how to use the compiler under CMS are contained in the DOS PL/I Optimizing Compiler: CMS User's Guide.

CHAPTER 3. HOW TO DEFINE A DATA SET

A data set is any collection of data in auxiliary storage that can be created or accessed by a program. It can be punched onto cards or paper tape; or it can be recorded on magnetic tape or on a direct-access storage device such as a magnetic disk. A printed listing can also be termed a data set, but it cannot be read by a program.

The language reference manual for this compiler describes how to use PL/I statements to transmit data between a program and a data set. For a program containing such statements to be executed, it is necessary to give the operating system certain information about the data set. Some of this information is placed in the file declaration in the PL/I source program; more may need to be given in DOS job control statements. The DOS job control statements that may be required to define data sets are described in _DOS/VSE System Control Statements_.

This chapter provides a short explanation of what is required for defining consecutive data sets.

DOS INPUT/OUTPUT CONCEPTS

Before a PL/I program can process a data set, it must identify the device that will read or write the data set. The Disk Operating System uses _symbolic device names_ rather than actual device addresses to identify input/output devices. Therefore, to relate a file to the device that will process the associated data set, the file declaration must specify the device type and a symbolic device name in the MEDIUM option of the ENVIRONMENT attribute. How this is done is explained in Chapter 7.

Each symbolic device name used in a DOS installation must be related to an actual unit address. Many such relationships are established permanently for an installation during initial program load; these are known as _standard device assignments_. Before you run a program, determine what the standard assignments are at your installation.

If you always use these standard assignments, it will be sufficient to identify the device that will process a data set in the MEDIUM option of the associated file. If you want to use a symbolic device name for which no permanent assignment has been made at your installation, or if you want to temporarily change an assignment, you must include an ASSGN statement in the job (see below).

PROCESSING A DATA SET

The amount of information that you must supply to the operating system when you run a program that creates or accesses a data set depends on the type of input/output device used and on whether you are using the standard device assignments. For each data set, the PL/I source program must have a file declaration that includes the MEDIUM option and specifies the record format for the data set. In addition, DOS job control statements must be supplied as follows:

- For each data set on labeled magnetic tape: a TLBL statement.
- For each data set on a diskette or a direct-access device: a DLBL statement and at least one EXTENT statement.
- For each nonstandard device assignment: an ASSGN statement.

- For one or more data sets that are either REGIONAL or INDEXED or held on labeled magnetic tape, a single LBLTYP statement must be supplied in the link-editing step for the program, dependent upon the level of your operating system.

Note: A LBLTYP statement is not required for VSAM data sets.

These requirements are summarized in Figure 2. The following paragraphs contain short descriptions of the job control statements and indicate where they are placed in the job stream.

All the job control statements for a job step must precede the EXEC statement for that job step.

Always Required For	**Information**	**Where Specified**
Any input/output	Type of device—unless device independent Symbolic device name Record format	File declaration in source program: see <u>OS and DOS PL/I Language Reference Manual</u>
Nonstandard device assignment	Device assignment	ASSGN statement
Data set on magnetic tape with standard labels	Identification Storage for label processing	TLBL statement LBLTYP statement
Data set on direct-access volume or on a diskette	Identification and extent information REGIONAL (1) REGIONAL (3) or INDEXED file, storage for label processing	DLBL statement EXTENT statement LBLTYP statement (not required for VSAM data sets, or for DOS/VSE with Advanced Functions)

Figure 2. Data Set Information that Must Be Supplied

ASSGN Statement

The ASSGN statement associates a symbolic device name with the address of an actual input/output device.

The ASSGN statement and the symbolic device names associated with its use are described in <u>DOS/VSE System Control Statements</u>.

The ASSGN statement can appear anywhere among the job control statements for the job step, other than between DLBL and EXTENT statements.

TLBL Statement

The TLBL statement applies only to data sets on magnetic tape. It contains information that identifies the data set (for example, the data set name and serial number). The operating system records this information on the tape in a series of records termed a <u>data set label</u>, and refers to it if the data set is used again in another job to ensure that the correct tape has been mounted.

The TLBL statement can appear anywhere among the job control statements for the job step, other than between DLBL and EXTENT statements.

LBLTYP Statement

A LBLTYP statement is not required for consecutive data sets on direct-access devices or for VSAM data sets; it is always needed for INDEXED and REGIONAL direct-access data sets and for data sets on magnetic tape with IBM standard labels. The LBLTYP statement is not required when running on DOS/VSE with VSE Advanced Functions. The statement requests the linkage editor to allocate space in the executable program phase for use by the operating system label-processing routines.

Dependent upon the level of your operating system, one LBLTYP statement must be included in any job that employs labeled magnetic tape. It must precede the EXEC LNKEDT statement.

DLBL Statement

The DLBL statement applies only to data sets on diskettes and direct-access devices. Like the TLBL statement for magnetic-tape data sets, it provides information that enables the operating system to write data set labels or to check them if an existing data set is to be processed.

The DLBL statement can appear anywhere among the job control statements for the job step. A DLBL statement must be followed by at least one EXTENT statement (see below).

EXTENT Statement

For a direct-access device an EXTENT statement defines the space (extent) to be occupied by a data set (that is, it identifies the actual tracks to be used). It is possible for a data set to extend over two or more extents (groups of contiguous tracks) of a direct-access device. In such a case, an EXTENT statement is required for each extent.

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

The EXTENT statements for a data set must always follow immediately behind the DLBL statement for the data set.

STANDARD FILES

PL/I includes two standard files, SYSIN for input and SYSPRINT for output. If the PL/I program includes a GET statement without the FILE option, the compiler assumes the file to be SYSPRINT associated with the symbolic device name SYSIPT. If the program includes a PUT statement without the FILE option, the compiler assumes the file to be SYSPRINT associated with the symbolic device name SYSLST. If neither SYSPRINT nor SYSIN is declared explicitly in the program, the compiler assumes the following declaration for SYSPRINT:

```
DECLARE SYSPRINT FILE PRINT
ENVIRONMENT(MEDIUM(SYSLST) F RECSIZE(121));
```

and, for SYSIN:

```
DECLARE SYSIN FILE STREAM INPUT
ENVIRONMENT(MEDIUM(SYSIPT) F RECSIZE(80));
```

EXAMPLES

The examples in Figure 3 on page 12 and Figure 4 on page 13 illustrate the use of the standard files SYSIN and SYSPRINT for punched-card input and printed output, the creation of a data set on magnetic tape, and the creation and access of a data set on a direct-access storage device. Each example includes the

DOS job control statements necessary for compiling, link-editing, and executing the program.

Figure 3 shows a program that evaluates the familiar expression for the roots of a quadratic equation and records its results in two data sets, one on magnetic tape and the other on an IBM 3330 Disk Storage drive.

```
// JOB FIG0302
// PAUSE PLEASE MOUNT TAPE 061699 AS '181'
// OPTION LINK
// EXEC PLIOPT, SIZE=64K
  CREATE:PROC OPTIONS(MAIN);
         DCL RESULTS FILE RECORD OUTPUT SEQL ENV(MEDIUM
              (SYS009,3330)FB BLKSIZE(400) RECSIZE(40),
         RESULT2 FILE STREAM OUTPUT
                 ENV(F RECSIZE(40) MEDIUM(SYS008,2400)),
         1 RECORD,2 (A,B,C,X1,X2) FLOAT DEC(6) COMPLEX;
         ON ENDFILE (SYSIN) GOTO FINISH;
         OPEN FILE (RESULT2),FILE(RESULTS),FILE(SYSPRINT),
              FILE(SYSIN);
  NEXT:  GET FILE (SYSIN) LIST (A,B,C);
         X1=(-B+SQRT(B**2-4*A*C)/(2*A);
         X2=(-B-SQRT(B**2-4*A*C0)/(2*A);
         PUT SKIP FILE (SYSPRINT) EDIT (RECORD) (C(E(16,9)));
         PUT FILE (RESULT2) EDIT (RECORD) (C(E(16,9)));
         WRITE FILE (RESULTS) FROM (RECORD);
         GOTO NEXT;
  FINISH:CLOSE FILE(RESULTS),FILE(RESULT2),FILE(SYSPRINT),
              FILE(SYSIN);
         END CREATE;

/*
// LBLTYP TAPE
// EXEC LNKEDT
// ASSGN SYS009,3330,VOL=DOS222,SHR
// ASSGN SYS008,X'181',X'C8'
// DLBL RESULTS,'ROOTS',SD
// EXTENT SYS009,DOS222,1,3458,19
// TLBL RESULT2,'RESULTS',0,061699
// EXEC, SIZE=64K
5 12 4
4 -10 4
5 16 2
4 -12 10
5 12 9
29 -20 4
/*
/&
```

Figure 3. Example of a Program that Creates a Data Set

The 3330 data set is written on disk pack serial number DOS222 (specified in the EXTENT statement); the DLBL statement specifies the data set name ROOTS, the default retention period of seven days, and the code SD for CONSECUTIVE organization.

The magnetic tape storage set is written on a tape mounted on the tape drive with address X'181' (second ASSGN statement); the TLBL statement specifies the data set name and the volume serial number of the magnetic-tape volume to be used.

Figure 4 shows a program that reads the 3330 data set created in the first example and prints the results. Although a different filename is used from that in the first example, the data set is identified by the name ROOTS, which is specified in the DLBL statement.

```
// JOB FIG0303
// OPTION LINK
// EXEC PLIOPT,SIZE=64K
  ACCESS:PROC OPTIONS(MAIN);
         DCL INDATA FILE RECORD INPUT SEQL ENV(MEDIUM(SYS009,3330)
             FB BLKSIZE(400) RECSIZE(40)),
               1 RECORD, 2 (A,B,C,X1,X2) FLOAT DEC(6) CPLX;
         ON ENDFILE (INDATA) GOTO FINISH;
         PUT EDIT ('A','B','C','X1','X2')
                  (X(7),3(A,X(23)),A,X(22),A);
         OPEN FILE (INDATA),FILE(SYSPRINT)
  NEXT:  READ FILE (INDATA) INTO (RECORD);
         PUT SKIP EDIT (RECORD)(C(F(12,2)));
         GOTO NEXT;
  FINISH:CLOSE FILE(INDATA),FILE(SYSPRINT);
          END ACCESS;
/*
// EXEC LINKEDT
// ASSGN SYS009,3330,VOL=DOS222,SHR
// DLBL INDATA,'ROOTS'
// EXTENT SYS009,DOS222
// EXEC, SIZE=64K
/&
```

Figure 4. Example of a Program that Accesses a Data Set

CHAPTER 4. THE OPTIMIZING COMPILER

The DOS Optimizing Compiler translates PL/I source statements into machine instructions. A set of machine instructions such as is produced by the compiler is termed an object module. (If several sets of PL/I source statements are present, each corresponding to an external procedure and separated by appropriate control statements, 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 standard routines such as those that handle the allocation of main storage space and the transmission of data between main and auxiliary storage, the compiler inserts references to standard subroutines that are stored in the DOS PL/I Resident Library.

An object module produced by the compiler is not ready for execution until the appropriate modules from the resident library have been included. This is the task of a system service program, the linkage editor, which is described in Chapter 5 on page 43. A module that has been processed by the linkage editor is referred to as an executable program, and sometimes as an executable program phase.

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

While it is processing a PL/I source program, the compiler produces a listing that contains information about the source program and the object module derived from it, together with diagnostic messages relating to errors or other conditions detected during compilation. Much of this information is optional and is supplied either by default or in response to a request made by including appropriate options in the compiler-control PROCESS statement that optionally precedes each source module.

The compiler also includes a facility, the processor, which can modify the source statements or insert additional source statements before compilation begins.

Compiler options 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, and the compiler can be used merely to check the syntax of the source-program statements. Furthermore, continuation of processing through syntax checking and compilation can be made conditional on successful compile-time processing. The compiler options are discussed under "Compiler Options" on page 17.

COMPILATION

The compiler comprises a control phase that remains in main storage throughout compilation, and a series of processing phases that are loaded and executed in turn under the supervision of the control phase. Several of the processing phases are loaded only if required for a particular compilation.

The source program must be in the form of a data set read by a device assigned to SYSIPT; frequently, the data set is a deck of punched cards. The source program is passed to the compiler either directly or through a preprocessor stage.

The source program may also be passed directly to the compiler if it contains %INCLUDE statements but no other type of preprocessor statements. In this case, you must specify the INCLUDE option.

The compiler translates the source program into machine instructions, and creates the external symbol dictionary (ESD) and relocation dictionary (RLD) required by the linkage editor. The external symbol dictionary is a list that includes the names of subroutines that are referred to in the object module but are not part of the module; these names, which are termed external references, include the names of the resident library modules and other object modules that will constitute the executable program. The relocation dictionary contains information that enables the linkage editor to assign absolute storage addresses within the object module. Chapter 5 contains a fuller discussion of the external symbol dictionary and the relocation dictionary, and explains how the linkage editor uses them.

JOB CONTROL FOR COMPILATION

Compilation is initiated by the following EXEC statement:

```
// EXEC PLIOPT,SIZE=nK
```

The optimizing compiler can be used in a batched-job foreground partition provided that a copy of the compiler was link-edited and cataloged into a private core image library for use in that particular partition when the system was generated, and provided that the symbolic device name SYSCLB is assigned to the partition before compilation is initiated. The volume containing the private core image library must be mounted on the device associated with the symbolic device name SYSCLB. (Under VSE/Advanced Functions, LIBDEF may be used.)

If the compilation job step is to be followed by a link-editing job step, the EXEC statement must be preceded by an OPTION statement specifying the LINK option, thus:

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

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

The compiler requires several standard data sets. These are shown in Figure 5 on page 16 and described in the following paragraphs.

Primary Input (SYSIPT)

The primary input to the compiler must be a consecutive data set containing a PL/I source module in the form of 80-byte unblocked records. The source statements may be preceded by a compiler-control PROCESS statement. This statement is used to specify the compiler options required for the compilation. The source module may comprise one or more external procedures; if you want to compile more than one external procedure in a single job step, separate the external procedures in the input data set with PROCESS statements. (This use of the PROCESS statement is described under "Batched Compilations" on page 38.)

The input data set may be on a diskette, direct-access device, punched cards, or magnetic tape. The address of the device used must be assigned to SYSIPT. The data set must contain unblocked fixed-length 80-byte records.

Function	Symbolic Name	Device Type	File	When Required
Input	SYSIPT	DASD Magnetic tape Card reader Diskette	IJSYSIN	Always
Listing	SYSLST	DASD Magnetic Tape Printer Diskette	IJSYSLS	Always
Output to linkage editor	SYSLNK	DASD	IJSYSLN	When link-editing follows compilation in the same job
Output to linkage editor (card deck)	SYSPCH	DASD Magnetic tape Card punch Diskette	IJSYSPH	When link-editing takes place in a subsequent job
Compiler spill files	SYS001 SYS002	Disk	IJSYS01 IJSYS02	Always
Source statement library	SYSSLB (if the source statement module is held in a private source statement library	DASD	IJSYSSL	When preprocessor %INCLUDE is used

Figure 5. Compiler Data Sets

Output (SYSLNK or SYSPCH)

The compiler can optionally transmit the object module to a data set on SYSLNK, to a data set on SYSPCH, or to a data set on both. The object module is in a form suitable for processing by the DOS linkage editor program.

Workspace (SYS001 and SYS002)

The compiler will require data sets on SYS001 and SYS002 for use as intermediate (temporary) workspace.

SYS001 and SYS002 define data sets, known as _spill files_, which the compiler uses for auxiliary storage during the compilation. These data sets must be on similar disk storage devices.

Optimal compilation speed is achieved if SYS001 and SYS002 are on different volumes with full cylinders allocated to each data set. If only one volume is available, SYS001 and SYS002 should use a split-cylinder extent allocation with the cylinders divided equally between the data sets.

Listing (SYSLST)

The compiler can produce a listing giving information about the program. The information that may appear, and the associated options, are described under "Listings" on page 30. The symbolic listing is SYSLST. Records associated with the listing are fixed-length, 121-byte records including an American National Standard carriage control character.

Source Statement Library

The preprocessor %INCLUDE statement can be used to obtain source statements for the program from the system source statement library or from a private source statement library. SYSSLB or LIBDEF under VSE/Advanced Functions must be assigned when a private source statement library is to be searched for the required source statement book.

COMPILER OPTIONS

The optimizing compiler offers a number of options that you can select by including the appropriate keywords in the *PROCESS statement. The options control various aspects of the program that will be generated, such as the extent to which it will be optimized, the contents of the listing, and other factors. Thus the options can be used to tailor the compiler to suit your needs.

The *PROCESS statement precedes the source statements in the input to the compilation step. The format of the statement is:

```
*PROCESS [option-list];
```

The * must appear in the first position of the statement. The keyword PROCESS follows the asterisk, with or without intervening blanks. The option-list follows the keyword *PROCESS with one or more intervening blanks. The options in the list are separated from each other by a comma or one or more blanks, or both. The statement must be terminated by a semicolon. An example is given below:

```
*PROCESS SIZE (72K),LIST,DECK;
```

A *PROCESS statement can extend over more than one input record, provided that the default right-hand margin is observed for each record. An option keyword may span two adjacent records if 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.

Many of the option keywords have an abbreviated form that you can use to obtain a more concise list of options. You may specify the options in any order.

The compiler options are of the following types:

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

2. Keywords that permit you to provide a value-list that qualifies the option (for example, SIZE(56K)).

3. A combination of 1 and 2 above.

For each compilation, a default for each option (except NAME, CATALOG, and CONTROL) will apply, unless specifically overridden by a request for a variant of the option.

Compiler Option	Abbreviated Name	IBM Default
AGGREGATE|NOAGGREGATE	AG|NAG	NOAGGREGATE
ATTRIBUTES[(FULL|SHORT)]| NOATTRIBUTES	A[(F|S)]|NA	NOATTRIBUTES Default suboption FULL
CATALOG('name')	—	—
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
DUMP|NODUMP	DU|NDU	NODUMP
DYNBUF|NODYNBUF	—	NODYNBUF
ESD|NOESD	—	NOESD
FLAG[(I|W|E|S)]	F[(W|E|S)]	FLAG(I)
FLOW[(n,m)]|NOFLOW	—	NOFLOW
GOSTMT|NOGOSTMT	GS|NGS	NOGOSTMT
GRAPHIC|NOGRAPHIC	—	NOGRAPHIC
INCLUDE|NOINCLUDE	INC|NINC	NOINCLUDE
INSOURCE|NOINSOURCE	IS|NIS	INSOURCE
LIMSCONV|NOLIMSCONV	LCS|NLCS	NOLIMSCONV
LINECOUNT(n)	LC(n)	LINECOUNT(55)
LINK|NOLINK[(W|E|S)]	—	NOLINK(S)
LIST[(m[,n])]|NOLIST	—	NOLIST
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)
MDECK|NOMDECK	MD|NMD	NOMDECK
NAME('name,origin[,NOAUTO]')	N('name,origin[,NOAUTO]')	—
NEST|NONEST	—	NONEST
OFFSET|NOOFFSET	OF|NOF	NOOFFSET
OPTIMIZE(TIME|0|2)|NOOPTIMIZE	OPT(TIME|0|2)|NOPT	NOOPTIMIZE
OPTIONS|NOOPTIONS	OP|NOP	OPTIONS
SIZE(yyyyyy|nnnnK|MAX)	SZ(yyyyyy|nnnK|MAX)	SIZE(MAX)
SOURCE|NOSOURCE	S|NS	SOURCE
STORAGE|NOSTORAGE	STG|NSTG	NOSTORAGE
SYNTAX|NOSYNTAX[(W|E|S)]	SYN|NSYN[(W|E|S)]	NOSYNTAX(S)
WORKFILE(2311|2314|3330| 3340|3350|FBA)[1]	—	WORKFILE(2311)
XREF[(FULL|SHORT)]|NOXREF	X[(F|S])|NX	NOXREF Default suboption FULL

[1]Users of the 3344 direct-access device should specify 3340 in the WORKFILE option.
Users of the 3375 or 3380 direct-access device should specify 3330 in the WORKFILE option.

Figure 6. Compiler Options, Abbreviations, and Defaults

Figure 6 lists all the compiler options alphabetically with their abbreviated forms and their default values. Figure 7 on page 19 lists them by function so you can more easily see the types of facilities that are available. The defaults given are those supplied by IBM. An installation can modify the defaults according to local requirements. Check for any modified defaults at your installation. It is possible for compiler options to have been deleted from use when the system was generated. If a deleted option is requested in a PROCESS statement, a message will be printed and the compilation will proceed without the use of the option. A deleted option can be restored for use temporarily if, when the option is specified, the CONTROL option is also specified. The CONTROL option is described later in this chapter.

The following paragraphs describe the options. For those options that request the compiler to list information, only a brief description is included; the generated listings are described under "Listings" on page 30.

COMPILER OPTIONS: FUNCTIONAL SUMMARY, PART I	
Control Listings Produced	
AGGREGATE	Lists aggregates and their sizes
ATTRIBUTES	Lists attributes of identifiers
ESD	Lists external symbol dictionary
FLAG(I|W|E|S)	Suppresses diagnostic messages below a certain severity.
INSOURCE	Lists preprocessor input
LIST	Lists compiled code produced by compiler
MAP	Lists offsets of STATIC INTERNAL and AUTOMATIC variables
OPTIONS	Lists options used
SOURCE	Lists source program or preprocessor output
STORAGE	Lists storage used
XREF	Lists identifiers and the statements in which they are 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
Define character set and margins of input	
CHARSET	Identifies a character set used in source
GRAPHIC	Specifies that graphics are used in source
MARGINS	Identifies the columns used for source program, and identifies the position of a carriage control character
Prevent unnecessary processing	
NOSYNTAX(W|E|S)	Stops processing after errors are found in preprocessing
NOCOMPILE(W|E|S)	Stops processing after errors are found in syntax checking
Control preprocessing	
MACRO	Allows full use of the preprocessor facility
INCLUDE	Allows inclusion of text without overheads incurred by macro
MDECK	Produces a source deck from preprocessor output
INSOURCE	Lists the input to the preprocessor

Figure 7 (Part 1 of 2). Compiler Options Arranged by Function

COMPILER OPTIONS: FUNCTIONAL SUMMARY, PART II	
Control executable program phase	
DECK	Produces an object module
CATALOG	Produces an object deck on virtual card punch with CATALR card
NAME	Specifies non-default name for executable program phase
Control storage used during compilation	
SIZE	Controls the amount of storage the compiler uses
Reduce execution-time storage	
DYNBUF	Allocates buffer space during execution
LIMSCONV	Specifies that certain conversions will not be used in stream I/O, consequently reducing number of library modules link-edited
Identify statement numbers	
GOSTMT	Specifies that a statement number table will be retained till execution time so that execution time messages can include statement number
OFFSET	Specifies that a listing associating statement numbers with offsets will be generated to aid in identifying statements from offsets given in execution time error messages
Use when debugging	
FLOW	Generates code so that a trace of executed statements will be retained
COUNT	Generates code so that a count of the number of times each statement is executed will be printed at the end of the program
Improve compilation/execution speed	
OPTIMIZE(TIME)	Reduces execution time at the expense of compilation
NOOPTIMIZE	Reduces compilation time at the expense of execution
Use when debugging the compiler	
DUMP	Produces a dump if the compiler itself terminates abnormally
Specify devices to be used by the compiler	
WORKFILE(device type)	Specifies device type used for compiler workfiles
Use for system programming	
CONTROL('password')	Allows access to deleted options for those who know password

Figure 7 (Part 2 of 2). Compiler Options Arranged by Function

AGGREGATE Option

An aggregate length table, giving the lengths and bytes of all major structures and arrays in the source program, will be produced if the AGGREGATE option is specified.

ATTRIBUTES[(FULL|SHORT)] Option

The ATTRIBUTES option requests the printing of a table of source program identifiers and their attributes.

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

CATALOG Option

The CATALOG option requests that the object module be stored and cataloged in a relocatable library at the end of the compilation step. It causes the compiler to generate a CATALR statement preceding the object module output on SYSPCH. The CATALOG option specifies the name by which the object module is to be identified in the library. The name can be from one to eight characters, the first of which must <u>not</u> be an asterisk. It must be enclosed between quotes. Further information about the use of this option is given in Chapter 6.

CHARSET Option

60- OR 48-CHARACTER SET: If the PL/I source statements are written in the PL/I 60-character set, specify CHARSET(60); if they are written in the 48-character set, specify CHARSET(48). The <u>OS and DOS PL/I Language Reference Manual</u> lists the character sets. (Note that the compiler will accept source programs written in either character set if CHARSET(48) is specified. However, the use of CHARSET(48) will cause an increase in compilation time.)

BCD OR EBCDIC: The compiler will accept source statements in which the characters are represented by either of two codes; binary coded decimal (BCD) or extended binary-coded-decimal interchange code (EBCDIC). The <u>OS and DOS PL/I Language Reference Manual</u> lists the EBCDIC representation of both the 48-character set and the 60-character set.

If both arguments (48 or 60, EBCDIC or BCD) appear, 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 might be produced; the other listings that follow the source program will not be produced.

CONTROL Option

The CONTROL option enables the compiler options deleted at system generation to be used for a particular compilation. The CONTROL option must be specified with a password that is defined at system generation.

Note: The CONTROL option must be the first in the list of options in the PROCESS statement.

If the CONTROL option is specified without a password, CONTROL ('OPTIMIZE') is defaulted.

COUNT Option

The COUNT option specifies that the compiled program is to produce a table indicating how many times each statement or group of statements in the program has been executed. The table is written to SYSLST when the program terminates. The counting is done within your MAIN procedure and inner procedures compiled with it. Any statements eliminated by optimization of the program are listed as "unexecuted statements." You can supply a head to identify the COUNT output by use of the PLIXHD facility described at the end of this section.

The COUNT option implies the GOSTMT option. If COUNT and NOGOSTMT are both specified, a diagnostic message is issued and no count table is produced.

DECK Option

The DECK option specifies that the compiler is to write the object module in the form of 80-position records onto SYSPH. Positions 73 through 76 of each record 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 module. Positions 77 through 80 contain a 4-digit decimal number: the first record is numbered 0001, the second 0002, etc.

DUMP Option

The DUMP option, when specified, causes a dump of registers and main storage used by the optimizing compiler if compilation terminates because of an error within the compiler itself.

DYNBUF Option

The DYNBUF option specifies that the compiler is not to allocate buffers for files at compile-time. Instead, the buffers are to be allocated dynamically when the files are opened at execution-time. Consequently, space is not required for a file's buffers until it is open and is released when it is closed. NODYNBUF is the default. NODYNBUF causes the compiler to allocate storage for buffers for all files, with the effect that the overall storage requirement of the object program is increased but the time taken to open the files is reduced.

If DYNBUF is used, there is no advantage in having all the files in a program open concurrently. Therefore, files that need not be opened concurrently should be opened and closed separately.

If NODYNBUF is used, there is no advantage in opening and closing files separately so that they are not open concurrently. Therefore, all the files used in a program can be opened and closed together.

ESD Option

The ESD option requests the inclusion of a listing of the external symbol dictionary (ESD).

FLAG Option

The diagnostic messages produced by the optimizing compiler are graded in order of severity. The FLAG option specifies the minimum level of severity that requires a message to be printed:

FLAG(I)	List all diagnostic messages. Note that, if you specify FLAG, FLAG (I) is assumed.
FLAG(W)	List all diagnostic messages except "informatory" messages.
FLAG(E)	List all diagnostic messages except "warning" and "informatory" messages.
FLAG(S)	List only "severe" errors and "unrecoverable" errors.

The severity levels are discussed under "Listings" on page 30.

FLOW Option

The FLOW option requests that the compiled program list the numbers of the last "n" branch-out and branch-in source statements executed prior to the occurrence of an interrupt that results in an execution-time diagnostic message. The format of the option is:

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

where

n	is the number of statement numbers to be listed, and
m	is the number of procedures through which a flow-trace is to be maintained at any one time.

The maximum value for n or m is 32,767. The FLOW option is operative only within your MAIN procedure and inner procedures compiled with it. The FLOW option is discussed further under "Statement Numbers and Tracing" on page 207.

If the FLOW option is specified without arguments, FLOW (25,10) is assumed.

GOSTMT Option

The GOSTMT option requests the compiler to produce additional information that will allow statement numbers from the source program to be included in diagnostic messages produced during execution of the compiled program.

However, you can get information about statement numbers and their associated offsets by referring to the "Statement Offset Addresses" on page 34.

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.

INCLUDE Option

The INCLUDE option requests the compiler to handle the inclusion of PL/I source statement books for programs that use the %INCLUDE statement. This method is faster than using the preprocessor for programs that contain %INCLUDE statements but no other preprocessor statements. The INCLUDE option should not be used if the MACRO option is specified.

INSOURCE Option

The INSOURCE option requests a listing of the PL/I source statements by the preprocessor.

LIMSCONV Option

The LIMSCONV option specifies that the compiler will not have to handle any conversions for data- or list-directed input other than the following:

- Bit (or character containing bit string) to bit.
- Character to character (or picture character).
- Fixed- or floating-point decimal constants (or character strings that represent such constants) to arithmetic.

The use of this option will result in a space saving. The resident library conversion modules for all other conversions are otherwise incorporated into the object module on the assumption that they might be used. Note that if a program attempts a conversion not given above when this option has been used, the CONVERSION condition will be raised. On-codes that indicate attempts to use suppressed conversions are given; these are listed in the OS and DOS Language Reference Manual.

LINECOUNT Option

The LINECOUNT option specifies the number of lines to be included in each page of a printed listing, including heading lines and blank lines. Its format is:

```
LINECOUNT(n)
```

where

n is the number of lines.

LINK Option

The LINK option specifies that link-editing is to follow the compilation unconditionally; the use of the NOLINK option will suppress link-editing according to the severity level of messages produced during the compilation. Note that the LINK option of the DOS OPTION statement is also needed if link-editing is to follow the compilation.

LIST Option

The LIST option requests a listing of the object module generated by the compiler (in a form similar to assembler language instructions). The format of the LIST option is:

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

where

m is the number of the first statement for which an object listing is required.

n is the number of the last statement for which an object listing is required.

If 'n' is omitted, an object listing for statement number 'm' only is given.

If LIST is used in conjunction with MAP, additional listings of static storage are generated.

MACRO Option

Specify MACRO when you want to employ the compile-time preprocessor. The use of the preprocessor is described under "Compile-Time Processing" on page 40.

MAP Option

The MAP option causes the printing of the tables showing the organization of the storage for the compiled object module. A table showing the mapping of static internal and automatic variable is always produced. This enables you to find variables in a PLIDUMP. If the LIST option is also used, maps of static internal and external control sections are also provided. They include a table showing the mapping of PL/I data items in dynamic and static storage. The MAP option is normally used in conjunction with the LIST option.

MARGINI Option

The MARGINI option defines the character that the compiler is to print on margins of the source listing, thus revealing any source statements that cross either margin. Its format is:

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

where

c is the alphameric character to be printed on the source listing margins.

MARGINS Option

The MARGINS (source margin) option specifies the part of each input record that contains the PL/I source statements. The compiler will not process data that is outside these limits. The option can also specify the position of an ANS carriage control character to format the listing of source statements produced by the compiler if the SOURCE option is specified. The format of the MARGINS option is:

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

where

m	represents the position in the input record of the first byte of the field that contains the 80-byte source statement record,
n	represents the position in the input record of the last byte of the source statement field, and
c	represents the position in the input record of the byte that will contain the control character.

The value m must be less than or equal to n, and neither must exceed 80. The value c must be outside the limits set by m and n. The valid control characters are:

b̲	Skip one line before printing (blank)
0	Skip two lines before printing
-	Skip three lines before printing
	Suppress space before printing
1	Start new page

Chapter 9 on page 118 contains a full description of the use of printer control characters. If you do not specify a position for a control character, it is assumed not to be used.

If the value c is greater than the maximum length of a source statement record, the compiler will not be able to recognize it; consequently the listing will not have the required format. If the character specified is not a valid control character, a blank is assumed by default.

If the value of m is 1 there is a possibility of confusion between source text and *PROCESS statement. If * PROCESS is found with the asterisk in column 1 it is taken as a * PROCESS statement even if it occurs in a source program. For this reason you should not set m to 1.

Source statements generated by the preprocessor always have a source margin (2,72). Columns 73 through 80 contain information inserted by the preprocessor; this information is described under "Listings" on page 30.

MDECK Option

Specify the option MDECK if you want the output from the preprocessor in the form of a card deck. This output is written (punched) as a data set on SYSPCH.

NAME Option

The NAME option specifies the name of the executable program phase that will be created by the linkage editor from the compiled object module. The option causes the compiler to place a linkage editor PHASE statement at the start of the object module. The PHASE statement has the effect of assigning the specified name and loading point address to the following module when the module is link-edited. The format of the NAME option is:

```
NAME('name,origin[,NOAUTO]')
```

where name, origin, and NOAUTO are the operands of the DOS linkage editor PHASE statement described under "PHASE Statement" on page 50.

NEST Option

The NEST option specifies that the source program listing should indicate for each statement its begin-block level and its DO-group level.

OFFSET Option

The OFFSET option causes printing of the statement numbers for statements internal to 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. Note that the GOSMT option will cause statement numbers, as well as offset addresses, to be included in execution-time diagnostic messages.

OPTIMIZE Option

The OPTIMIZE option specifies the type of optimization required:

> NOOPTIMIZE produces the fastest possible compilation, but inhibits optimization for faster execution and reduced storage requirements. The compiler still carries out optimization of the object code, but certain optional optimization is omitted.
>
> OPTIMIZE(TIME) requests the compiler to optimize the machine instructions generated for minimum execution time. A secondary effect of this type of optimization can be a reduction in the amount of storage required for object programs. 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 *OS and DOS PL/I Language Reference Manual* includes a full discussion of program optimization and efficient programming.

OPTIONS Option

The OPTIONS option requests a list showing the status of all the compiler options after any default attributes have been applied at the start of compilation.

SIZE Option

The optimizing compiler *must* have at least 51,200 (50K) bytes of main storage available for its use. The SIZE option specifies the amount of main storage available for the compilation. In a non-multiprogramming environment, the amount will be all of main storage other than that used by the supervisor. In a multiprogramming environment, the amount is limited to the size of the partition and the SIZE option on the EXEC statement. Code this option in one of the following ways:

> SIZE(yyyyyy) specifies that yyyyyy bytes of main storage are available for the compilation. Leading zeros are not required.
>
> SIZE(yyyK) specifies that yyyK bytes of main storage are available for the compilation (1K=1024). Leading zeros are not required.

SIZE(MAX) instructs the compiler to obtain as much main storage as it can.

Always use as much storage as possible to obtain maximum performance from the compiler. The point at which increasing the storage available does not significantly improve performance is determined mainly by the size of the source program. In general, the larger the source program, the greater the amount of main storage that will be necessary for maximum performance.

SOURCE Option

The SOURCE option requests a listing of the PL/I source statements processed by the compiler. The source statements listed are either those of the original source program or, if the MACRO option is specified, the output from the preprocessor.

STORAGE Option

The STORAGE option causes printing of a table giving the storage requirements for the compiled object module.

SYNTAX Option

The SYNTAX option specifies that the compiler will check the syntax of the source statements. If you specify NOSYNTAX, syntax checking will not be performed, and as a consequence, the program will not be compiled. (Note that, if you have specified SOURCE, a source listing will still be generated, even though the program is not compiled.)

If you specify the MACRO option, syntax checking can be conditional on the severity level of the diagnostic messages generated by the preprocessor:

NOSYNTAX(W)	No syntax checking if preprocessor issues a warning, error, or severe error message.
NOSYNTAX(E)	No syntax checking if the preprocessor issues an error or severe error message.
NOSYNTAX(S)	No syntax checking if the preprocessor issues a severe error message.

WORKFILE Option

The optimizing compiler always uses work files (or spill files) on a direct-access device for the temporary storage of text and/or dictionary information. The type of direct-access device that will normally be used is decided at system generation according to the resources of the installation. The WORKFILE option permits the selection of the alternative direct-access device for a particular compilation.

If the alternative device type is requested, the symbolic device names SYS001 and SYS002 must be assigned to the channel and the device(s) used, and DLBL and EXTENT statements must be provided in the job step for the compilation to define the data sets for each workfile (unless VSAM Space Management is used). For workfiles on fixed block devices, the CISIZE parameter on the DLBL statement must not be specified. The file names used in the DLBL statements must be IJSYS01 and IJSYS02. The amount of space required for the data sets is described in <u>DOS PL/I Optimizing Compiler: Installation Guide</u>.

When using the VSE/VSAM Space Management for SAM feature, SAM data sets may be defined in VSAM space. To use compile time workfiles in VSAM space, FBA (for fixed block architecture devices) must be specified. The data sets for the workfiles may be defined explicitly or implicitly as follows:

- For explicit defining, use Access Method Services to define a dynamic SAM ESDS with a default RECORDSIZE and a RECORDFORMAT of UNDEF. During compilation, supply DLBL statements for IJSYS01 and IJSYS02 specifying VSAM and DISP=(,DELETE). No EXTENT statement is needed.
- For implicit defining, supply DLBL statements during compilation for IJSYS01 and IJSYS02 specifying VSAM, DISP=(,DELETE) and the RECORDS and RECSIZE parameters. The volume may be specified either via an EXTENT statement or a default model for a SAM ESDS.

Optimum compilation speed is achieved if SYS001 and SYS002 are on different volumes with full cylinders allocated to each data set. If only one volume is available, SYS001 and SYS002 should use a split-cylinder extent allocation with the cylinders divided equally between the data sets.

The size and total number of records written by the compiler onto these data sets are listed at the end of the compilation; it varies widely according to the size and nature of the source program and the amount of main storage available. However, 250K bytes of storage for each data set should be sufficient for compiling programs containing up to 500 source statements.

If you are using an IBM 3344 direct-access device for workfiles, you must specify WORKFILE(3340). If you are using an IBM 3375 or 3380, you must specify WORKFILE(3330).

If you are using a fixed block device and the compiler is executing in a minimum size partition, the data set may not exceed 16,776K bytes. The limit on data set size is higher for larger partitions.

XREF[(SHORT|FULL)] Option

The XREF option specifies that the compiler is to include the cross-reference table of names used in the program, together with the numbers of the statements in which they are declared or referenced. Refer to the section "Cross-reference Table" on page 33 for a description of the format and content of the cross-reference table.

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.

Using PLIXHD to Identify COUNT Output

When COUNT output is generated, if your program contains a static external character variable called PLIXHD, the value in PLIXHD is printed at the head of the output. 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 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.

If PLIXHD is declared EXTERNAL but not CHARACTER VARYING, a diagnostic message is generated during compilation. If PLIXHD is EXTERNAL 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.

LISTINGS

During compilation, the compiler generates a listing that contains information about the compilation and about the source and object modules. It places this listing in the data set on SYSLST. The following description of the listing refers to its appearance on a printed page.

The listing comprises a small amount of standard information that always appears, together with those items of optional information requested or supplied by default. Figure 8 lists the optional components of the listing and the corresponding compiler options.

Listings	Options Required
Options for the compilation	OPTIONS
Preprocessor input	MACRO and INSOURCE
Source program	SOURCE
Statement nesting level	NEST
Attribute table	ATTRIBUTES
Cross-reference table	XREF
Aggregate-length table	AGGREGATE
Statement offset addresses	SOURCE, NOGOSTMT, and OFFSET
Storage requirements	STORAGE
External symbol dictionary	ESD
Static storage map	MAP
Object module	LIST
Diagnostic messages for severe errors, errors, warnings, and informatory conditions	FLAG(S), FLAG(E), FLAG(W), FLAG(I)

Figure 8. Compiler Listing and Associated Options

The first page of the compiler listing is identified by the compiler version number, the date, and, if the timer is used, the time.

All the pages of the listing are numbered sequentially in the top right-hand corner. Page 1 also includes a statement of the options specified for compilation.

The listing ends either with a statement that no messages were produced for the compilation, or with one or more diagnostic messages. The format of the messages is described under "Diagnostic Messages" on page 37. If the machine has the timer feature, the listing also ends with a statement of the elapsed time taken for the compilation.

The compiler also prints the number and size of the records written onto the spill files (IJSYS01 and IJSYS02). The information is provided to assist the system programmer in specifying the most economic amount of space for this file.

The following paragraphs describe the optional parts of the listing in the order in which they appear. Appendix A on page 281 includes a fully annotated example of a compiler listing.

OPTIONS USED FOR THE COMPILATION

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

PREPROCESSOR INPUT

If both the options MACRO and INSOURCE apply, the compiler lists the input statements to the preprocessor, one record per line. The printed lines are numbered sequentially at the left.

If the compiler detects an error, or the possibility of an error, during the preprocessor phase, it prints a message on the page or pages following the input listing. The format and classification of the error messages are exactly as described for the compilation error messages, under "Diagnostic Messages" on page 37.

SOURCE PROGRAM

If the option SOURCE applies, the compiler lists the source program input, one record per line; if the input statements include printer control characters, the lines will be spaced accordingly.

Additional formatting of both the preprocessor and compiler source listing is made possible by the %SKIP, %PAGE, %PRINT, and %NOPRINT, listing control statements. The %SKIP control statement causes the number of blank lines specified as its argument to be inserted. For example, the statement:

```
%SKIP(10) ;
```

will cause ten blank lines to be inserted into the source listing. The %PAGE statement causes the source program listing to be interrupted and resumed on the following page.

If a %NOPRINT statement is found, printing of a listing stops until a %PRINT statement is reached. The %PRINT and %NOPRINT statements are printed on the source listing, if PRINT is in effect when they are found.

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 were generated by the preprocessor, columns 73 through 84 contain the following information:

COLUMN	INFORMATION
73 - 80	Input line number from which the source statement was generated. This number normally corresponds to the line number in the preprocessor input listing. The line numbers will not correspond if MARGINS(m-n) is specified such that m-n¬=70.
81	Blank

82, 83	Two-digit number giving the maximum depth of replacement by the preprocessor for this line. If no replacement occurred, the columns are blank.
84	"E" signifies that an error occurred while replacement was being attempted. If no error occurred, the column is blank.

Statement Nesting Level

If the options SOURCE and NEST apply, the block level and the DO level are printed to the right side of the statement number under appropriate headings:

```
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 program together with their declared and default attributes. If the option XREF applies, the compiler prints a cross-reference table containing a list of the identifiers in the program together with the numbers of the statements in which they appear. If both ATTRIBUTES and XREF apply, the two tables are combined.

The suboption SHORT can be used to prevent unreferenced identifiers being listed. For details, see the earlier descriptions of ATTRIBUTES and XREF.

Attribute Table

If an identifier was declared explicitly, the number of the DECLARE statement is listed under the heading of DCL NO. If an identifier is declared contextually or implicitly, the symbol "******" is printed under this heading. The statement numbers of statement labels and entry labels are also given under this heading.

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 the expressions are replaced by asterisks.

For arrays or structures, the bounds are always shown as asterisks, even when they are declared as constants.

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

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 blocks 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 might appear only once within a source statement might acquire multiple references to the same statement number. By the same token, other names might appear to have incomplete listings of references, while still others might have references to statements in which the name does not appear explicitly. For example:

- Duplicate references might 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 for that name.
- Reference within DECLARE statements to variables that are not being declared are not listed. For example, given 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 elements 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.)

- The length of each element.

- The total length of each aggregate, structure, and sub-structure.

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

The statement number is the number of the DECLARE statement for the aggregate.

The length of an aggregate may not be known at compilation, either because the aggregate contains elements having adjustable extents, or because the aggregate is dynamically defined. In these cases, the word "ADJUSTABLE" or "DEFINED" appears in the "Length in Bytes" column.

If the program contains arrays or structures that are used in FORTRAN or COBOL programs, the compiler may have to create additional arrays and structures where mapping is different from that used in PL/I. Temporary arrays and structures created in this way will cause additional entries to be included in the table with the word "COBOL" or "FORTRAN" appended. A separate entry is made in the aggregate table for every aggregate dummy argument or FORTRAN-mapped array or COBOL-mapped structure.

STORAGE REQUIREMENTS

If the options SOURCE and STORAGE apply, 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 module that contains the executable part of the program.

- The length of the static internal control section. This control section includes storage for variables declared STATIC INTERNAL.

STATEMENT OFFSET ADDRESSES

If the options SOURCE, NOGOSTMT, and OFFSET apply, 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 within Procedure" 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 names in the message and then finding the largest entry in the table that is less than or equal to the offset in the message. If the procedures 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 the 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/);
    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
```

<u>Message</u>

```
IBM301I '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 (ESD)

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

The ESD is a table containing all the external symbols that appear in the module. The information appears under the following headings:

SYMBOL An 8 character field which identifies the external symbol.

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

SD Section definition: the name of a control section within this module.

CM Common area: a type of control section that contains no executable instructions. The compiler creates a common area for each non-string element variable declared STATIC EXTERNAL without the INITIAL attribute.

ER	External reference: an external symbol that is not defined in this module.
WX	Weak external reference: an external symbol that is not defined in this module and which is not to be resolved unless an ER entry is encountered for the same reference.
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: the entries in the ESD are numbered sequentially, starting from 0001.
ADDR	Hexadecimal representation of the address of the symbol: this field is not used by the compiler, since the address is not known at compile time.
LENGTH	The hexadecimal length in bytes of the control section (SD and CM entries only).

ESD Entries

The external symbol dictionary always starts with standard entries (Figure 9 on page 37):

1. Name of the initial entry point 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 the PL/I program calls another external procedure; in such a case, control passes directly to the entry point named in the CALL statement, and not to IBMBPIR.)

2. Name of the program control section (the control section that contains the executable instructions of the object module). This name is the first label of the PROCEDURE statement for the external procedure, padded on the left with asterisks to seven characters if necessary, and extended on the right with the character 1.

3. Name of the static internal control section (which includes storage for all variables declared STATIC INTERNAL). This name is the first label of the PROCEDURE statement for the external procedure, padded on the left with asterisks to seven characters if necessary, and extended on the right with the character 2.

4. External reference to entry point A in IBMBPIR, the PL/I initialization routine that establishes the PL/I environment for the object module. IBMBPIR is the entry point used when the PL/I program is invoked directly by means of a job control command or statement.

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

1. Section definition for the 4-byte control section PLIMAIN, which contains the address of the principal entry point to the external procedure. This control section is present only if the PROCEDURE statement includes the option MAIN. If it does not include the option MAIN, PLIMAIN is inserted as an external reference.

2. Weak external reference to PLITABS, PLIFLOW, and PLICOUNT: PLITABS in case a library module in a PL/I structure with the attributes STATIC and EXTERNAL is used to alter the

standard PL/I tab settings, PLIFLOW to enable the FLOW option to be used, and PLICOUNT to enable the COUNT option to be used.

3. LD-type entries for all names of entry points to the external procedure except the first.

4. ER-type entries for library routines and external routines called by the program. The list includes the names of library routines called directly by compiled code (first-level routines), and the names of routines that are called by first-level routines.

5. CM-type entries for variables declared STATIC EXTERNAL without the INITIAL attribute.

6. SD-type entries for all other external variables and for external file names.

7. WX-type entries for library routines and external routines that are not to be resolved unless ER-type entries for the same routines are present when the object module is link-edited.

Symbol	Type	ID	ADDR	Length
PLISTART	SD	0001	000000	000010
**PROG1	SD	0002	000000	0008F8
**PROG2	SD	0003	000000	000440
PLITABS	WX	0004	000000	
PLIFLOW	WX	0005	000000	
PLICOUNT	WX	0006	000000	
IBMPIRA	ER	0007	000000	
PLIMAIN	SD	0008	000000	000008

Figure 9. Standard ESD Entries

OBJECT MODULE LISTING

If the MAP option applies, the compiler generates a formatted listing of the contents of the static control section; this listing is termed the <u>static internal storage map</u>. The MAP option also produces a <u>variable storage map</u>. This map shows how PL/I data items are mapped in main storage. It names each PL/I identifier, its level, its offset (in both decimal and hexadecimal form) from the start of the storage area, its storage class, and the name of the PL/I block in which it is declared. The LIST option requests a listing of the machine instructions of the object program, including any compiler-generated subroutines, in a form similar to assembler language.

These listings contain information that cannot be fully understood without a knowledge of the structure of the object program as generated by the optimizing compiler. This is beyond the scope of this manual, but a full description of the object program and of the static storage map and the object-program listing can be found in the publication <u>DOS PL/I Optimizing Compiler: Execution Logic</u>.

Both the static internal storage map and the object program listings start on a new page.

DIAGNOSTIC MESSAGES

The compiler generates messages that describe any errors, or conditions that may lead to errors, detected during compilation. Messages generated by the preprocessor appear in the compiler listing immediately after the listing of the statements processed by the preprocessor; all other messages are grouped together at the end of the listing. The messages are graded according to their severity:

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

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

An error (E) message indicates an error for which the compiler has applied a "fix-up" with confidence. The resulting program will execute and will probably give correct results.

A severe (S) error message indicates an error that cannot be corrected with any degree of confidence by the compiler. Execution will almost certainly fail or produce incorrect results.

An unrecoverable (U) error message indicates an error that forces termination of the compilation.

You can generate your own messages to give information about preprocessing in the %NOTE statement. For example, you might to put out a message listing how many times a preprocessor procedure had been called.

The compiler lists only those messages with a severity equal to or greater than that specified by the FLAG option.

Type of Message	**Option**
informatory	FLAG(I)
warning	FLAG(W)
error	FLAG(E)
severe error	FLAG(S)
unrecoverable error	always listed

Each error message is identified by an 8-character code:

1. The first three characters are IEL, which identify all messages from the optimizing compiler.

2. The next four characters are a 4-digit message number.

3. The last character is the letter I, which is the operating system code for a message which does not require a response from an operator.

The text of each message, an explanation, and any recommended programmer response are given in DOS PL/I Optimizing Compiler: Messages.

BATCHED COMPILATIONS

The batched compilation facility allows the compiler to compile more than one external procedure in a single job step. Batch compilation can increase compiler throughput by reducing operating and compiler initialization overheads. An unrecoverable error during compilation of one external procedure will not usually prevent the compilation of those that follow it.

Examples of applications of batched processing are:

- Compiling a series of unrelated external procedures for debugging purposes

- Compiling a set of related external procedures to produce separate object modules that can be link-edited to form an overlay program (see Chapter 5).

To specify batched compilation, include a compiler PROCESS statement in front of each external procedure. The PROCESS statements identify the start of each external procedure and allow compiler options to be specified individually for each compilation. The format of the statement is described under "Compiler Options" on page 17.

USE OF SIZE AND DUMP OPTIONS

The following points should be noted when using the SIZE or DUMP option in a batched compilation:

1. The SIZE option specified either explicitly or by default for the first compilation in a batch cannot be altered for a subsequent compilation.
2. If the DUMP option is used in any compilation other than the first compilation in a batch, the option must be specified for the first compilation also.

EXAMPLE

Figure 10 is an example of simple batched compilations. It illustrates the use of a single invocation of the optimizing compiler to compile three procedures.

```
// JOB FIG0405X
// EXEC PLIOPT,SIZE=64K
* PROCESS A,X,SZ(50K),NEST,DECK,CATALOG('FIRST'),NOLINK;
  .
  .  First PL/I source module
  .
* PROCESS ESD,OPT(2),DECK,A,CATALOG('SECOND'),NOLINK;
  .
  .  Second PL/I source module
  .
* PROCESS A,X,CATALOG('THIRD'),DECK,NOLINK;
  .
  .  Third PL/I source module
  .
/*
/&
```

Figure 10. Example of Batched Compilations

The first PROCESS statement in Figure 10 specifies the options for the compilation of the procedure FIRST; of the options specified, only SIZE applies to the compilations of the other procedures. Each subsequent source module in the batched compilation is preceded by a PROCESS statement that specifies the options for that particular compilation. The use of the CATALOG and DECK options causes the compiler to produce a CATALR statement and an object module for each compilation on SYSPCH to facilitate the inclusion of each module in a relocatable library. See Chapter 6 for further information.

MULTIPLE COMPILATIONS IN A SINGLE JOB

In addition to the batched compilation facilities, it is also possible to reinvoke the optimizing compiler (and any other DOS language processor) to produce several object modules on SYSLNK. These modules may then be link-edited into an executable program and executed in single link-edit and execute steps. A sequence of job control statements to combine two PL/I procedures and assembler-language module as a single program follows:

```
// JOB
// OPTION LINK
// EXEC PLIOPT,SIZE=64K
        .
        .
        .
        first PL/I source module
        .
        .
        .
/*
// EXEC PLIOPT,SIZE=64K
        .
        .
        .
        second PL/I source module
        .
        .
        .
/*
// EXEC ASSEMBLY
        .
        .
        .
        Assembler-language module
        .
        .
        .
/*
// EXEC LNKEDT
// EXEC ,SIZE=64K
/&
```

COMPILE-TIME PROCESSING

The following discussion supplements the information contained in the OS and DOS PL/I Language Reference Manual by providing an illustration of how to invoke the preprocessor.

INVOKING THE PREPROCESSOR

The preprocessor stage of the compiler is executed only if you specify the compiler option MACRO.

The term MACRO owes its origin to the similarity of some applications of the compile-time facilities to the macro language available with such processors as the assembler. Such macro language allows you write a single instruction in a program to represent a sequence of instructions that have previously been defined.

Three other compiler options, MDECK, INSOURCE, and COMPILE are meaningful only when you also specify the MACRO option. All are described earlier in this chapter.

The job control statements and the PROCESS statement for a compilation step that uses the preprocessor are given below. The statements produce a source module and listings of both the original source module and the source module created by the preprocessor.

```
// JOB FIG0406X
// EXEC PLIOPT,SIZE=64K
* PROCESS MACRO,INSOURCE,MDECK;
   .
   .   PL/I source module
   .
/*
/&
```

THE %INCLUDE STATEMENT

The OS and DOS PL/I Language Reference Manual describes how to use the %INCLUDE statement to incorporate source statements from a source statement library into a PL/I source program. (A source statement library is used for the storage of sequences of source statements termed books.) Books that are related, such as the PL/I source statement books, are grouped together to form a sublibrary. Thus, a set of source statements that you might wish to insert into your source program by means of a %INCLUDE statement must exist as a book within a sublibrary within a library. Chapter 6 describes how to insert a book of source statements into a source statement library, and how to delete unwanted books.

The %INCLUDE statement specifies a list of one or more identifiers. Each specifies the name of the source statement book, or, optionally, a sublibrary and, in parentheses, the name of the book. For example, the statement:

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

specifies that the source statements in books INVERT and LOOPX contained in the sublibrary P and in the PL/I sublibrary are to be inserted consecutively into the source program generated by the preprocessor. If the books are held in a private source statement library, the symbolic device name SYSSLB must be assigned in an ASSGN statement preceding the compilation job step, unless you are operating under DOS/VSE with Advanced Functions, in which case the LIBDEF statement is used.

USE OF THE INCLUDE OPTION

If the %INCLUDE statement is the only preprocessor statement in the source program and the included source statements do not contain any other types of preprocessor statements, the INCLUDE compiler option can be specified instead of the MACRO option. This will result in a faster compilation.

Use of the %INCLUDE option will conflict with the use of INSOURCE and MDECK options as well as with the MACRO option. When the INCLUDE option is used, the limit of nesting of source statement books is eight. (There is no limit if the MACRO option is used.)

The listing of the source program produced when the INCLUDE option is used will show both the %INCLUDE statement and the source statements that replace it. Thus there is no need to use the MACRO and INSOURCE options to obtain a separate listing of the original source program and the included source statements.

The listing of a source program line that contains a %INCLUDE statement to include two source statement books is shown in Figure 11 on page 42.

Original source program line:

```
          X=Y; %INCLUDE R,W;   X=Z;
```

Appearance in the listing after expansion:

```
          X = Y; %INCLUDE R,********
                    .
                    .
                    .
                    Source statements for R
                    .
                    .
                    .
          ***************W;******
                    .
                    .
                    .
                    Source statements for W
                    .
                    .
                    .
            **********************X=Z;
```

Figure 11. Listing of Source Programs with the INCLUDE Option

Errors in the use of a %INCLUDE statement will be indicated at the appropriate point in the listing. Such errors are fully described by diagnostic messages.

CHAPTER 5. THE LINKAGE EDITOR

This chapter describes the use of the DOS linkage editor program. You must invoke the linkage editor program to process the object module compiled by the optimizing compiler before the PL/I program can be executed.

The linkage editor has two major functions. One is to resolve any unresolved machine addresses in the PL/I object module, to enable it to be loaded and executed in the required main storage partition; the other is to incorporate any required object modules from the relocatable library (such as modules from the PL/I resident library for operations for which the compiler does not generate object code). The resultant output from the linkage editor is an executable program phase. This output is written into a core-image library, from which it can be subsequently loaded for execution.

Input to the Linkage Editor

The linkage editor can accept input as follows:

- Linkage editor control statements from SYSIPT, SYSRDR, or SYSLNK. Linkage editor control statements can also be included in object modules loaded from a relocatable library.

- Relocatable object modules from:

  SYSLNK (compiler output)

  SYSRES (system relocatable library)

  SYSRLB (private relocatable libraries)

Output from the Linkage Editor

The linkage editor produces the following outputs:

- An executable program phase, either cataloged permanently or stored temporarily in a core-image library.

- A module map, and if necessary, diagnostic messages to indicate any detected error conditions, on SYSLST. The listing is described later in this chapter.

Additional Linkage Editor Processing

The linkage editor also has the following additional features:

- It can build an overlay structure for a program consisting of several procedures, so that programs having a total object-time storage requirement that exceeds the main storage available can be executed.

- Automatic library call feature (AUTOLINK), whereby PL/I library subroutines are automatically incorporated into the program phase by the linkage editor. This facility can be suppressed, either for a complete job-step, or for a particular phase in a multiphase link-edit operation.

OBJECT MODULE AND PROGRAM PHASE STRUCTURE

An object module produced by the optimizing compiler consists of:

- External symbol dictionary (ESD) entries
- Text (TXT) entries
- Relocation dictionary (RLD) entries
- END statement

Each executable program phase is constructed from the text of one or more object modules. The linkage editor constructs a control dictionary from the ESD information in each object module. A composite control dictionary is also constructed for RLD information. These dictionaries are used to resolve all linkages, both between external modules, and between any control sections generated by the compiler for a single source program module.

Text

The text of an object module consists of the compiler-generated machine instructions. These instructions are grouped in blocks known as *control sections*. An object module created by the optimizing compiler includes the following control sections:

- Program control section: contains the executable part of the program.
- Static internal control section: contains the following:
  - Constants used in the PL/I program
  - Addresses of static storage items
  - Addresses of library routines and compiler-generated subroutines
  - Addresses of other program control sections
  - Addresses of external entry points and external variables
  - STATIC INTERNAL variables
  - Control blocks used by the program, such as data element descriptors and skeleton control blocks for partially-completed argument lists
  - Addresses of label constants
- Control sections termed *common areas* are created for all external variables which, if not declared with the INITIAL attribute, will be common areas.
- PLIMAIN: this control section contains the addresses of the executable PL/I program. It is produced by the compiler only if the external procedure starts with a PROCEDURE statement having the MAIN option.
- PLISTART: this control section is always produced and is the entry point to the executable program. It contains the addresses of the entry points of library routine IBMBPIR and instructions to branch to this routine. Control is passed to this routine, which initializes the PL/I environment. Control is then passed to the address contained in the control section PLIMAIN. The address in PLIMAIN is resolved by the linkage editor. If two or more main external PL/I procedures are to be link-edited into one executable program, the linkage editor will place the address of the external entry point of the first procedure into PLIMAIN. If a mixture of PL/I and non-PL/I modules is to be link-edited to form a program in which a non-PL/I program is to receive initial control from the DOS supervisor, you must define the initial entry point explicitly to the linkage editor program in an ENTRY statement. The linkage editor ENTRY statement is described later in this chapter.

- PLIFLOW: this control section is produced if the statement-number trace is required (that is, if the FLOW option has been specified for the compilation). PLIFLOW invokes a module to initialize the flow-trace table that is maintained in static storage.
- PLICOUNT: this control section is produced if the statement-count is required (that is, if the COUNT option has been specified for the compilation). PLICOUNT invokes a module to initialize the count tables that are contained in non-LIFO (last-in-first-out) storage.
- Control sections for compiler-generated subroutines.
- External references to PL/I library modules to be link-edited into the program.

External Symbol Dictionary

The external symbol dictionary (ESD) contains a list of all the external symbols that appear in the module. An external symbol is a name that can be referenced 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 external entry names in the external procedure of a PL/I program. References to external symbols defined elsewhere are known as external references.

External references in a PL/I object module always include the names of the resident library modules that are required for the execution of the program. They may also include calls to relocatable subroutines that are not part of the PL/I resident library but which are to be incorporated into the executable program.

Part of the linkage editor's job is to locate the subroutines referred to, and include them in the program phase that is being link-edited. This is performed automatically by the AUTOLINK feature. If AUTOLINK is suppressed, any such module can be included if specifically requested by the use of the linkage editor INCLUDE statements. The INCLUDE statement is described under "Linkage Editor Control Statements" on page 48.

Relocation Dictionary

The relocation dictionary is built up from the RLD entries in each of the relocatable object modules that form the input to the linkage editor. It is used to establish the absolute machine addresses by adding compiler-generated addresses to the relocation factor used for a particular phase. The relocation factor is derived from the main storage location at which the phase is to be loaded.

At execution time, the machine instructions, including the instructions generated by the optimizing compiler, use two methods of addressing locations in main storage:

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

Object programs produced by the optimizing compiler are self-locating. The base addresses used within a phase are resolved by the linkage editor to the relative address.

LINKAGE EDITOR PROCESSING FOR A PL/I PROGRAM

An object module compiled by the optimizing compiler cannot be executed without the appropriate PL/I resident and transient library modules. The library modules are included in two ways:

- By incorporation during link-editing
- By dynamic invocation during execution

The first method is used for the resident library modules; the following paragraphs describe how the linkage editor locates those modules that are incorporated at link-edit time. The second method is used for the PL/I transient library subroutine modules associated with some input/output operations (including opening and closing files), and for execution-time error handling.

In its basic processing mode, the linkage editor accepts data from its primary input source, a data set with the symbolic device name SYSLNK. For a PL/I program, this input data set is the object module created by the compiler. The linkage editor uses the external symbol dictionary in the input module to determine whether the module includes any external references for which there are no corresponding external symbols within the module itself. It then attempts to resolve such references.

External symbol resolution by automatic library call involves a search of the relocatable library. The linkage editor locates the modules in which the external symbols are defined (if such modules exist), and incorporates them into the program phase it is creating. It takes each external reference it encounters and incorporates the corresponding module. If further identical references are encountered, no further action is taken other than to establish linkage to the incorporated module in each case.

Note that, if a PL/I and a FORTRAN object module have common STATIC EXTERNAL storage defined by means of a FORTRAN COMMON statement, the PL/I object module must be the first module processed by the linkage editor.

Multiprogramming Considerations

In a multiprogramming environment, you should know prior to the link-editing process whether the PL/I program is to be executed in the background partition or in one of the foreground partitions. A program is link-edited to be executed in a particular partition unless it is a self-relocating program.

Under DOS/VS, the relocating loader can load a program at any address in any partition without the need to write self-relocating programs or to link-edit again.

The use of the PHASE and ACTION statements is described in "Linkage Editor Control Statements" on page 48.

JOB CONTROL STATEMENTS FOR THE LINKAGE EDITOR

DOS job control statements associated with the use of the linkage editor program are:

LBLTYP statement

OPTION statement

EXEC statement

The LBLTYP and OPTION statements always precede the EXEC statement.

EXEC Statement

The linkage editor is invoked by the following statement:

```
// EXEC LNKEDT
```

LBLTYP Statement

The LBLTYP statement is used to define the amount of main storage to be reserved by the linkage editor in a program phase that processes either magnetic-tape data sets with IBM standard labels or non-sequential direct-access data sets. This storage is used for processing the labels of such data sets. It is required for both background and foreground programs, dependent upon the level of your operating system. It is not required for programs that process VSAM data sets or magnetic-tape data sets that are unlabeled or have non-standard tape labels. The format of the LBLTYP statement is:

```
// LBLTYP  [NSD(nn)|TAPE]
```

NSD (nn) is used if any non-sequential DSAD data sets are to be processed, regardless of other types of data sets to be used. 'nn' specifies the largest number of EXTENT statements that will be given for any of the DLBL statements that are associated with the job step.

TAPE is used only if magnetic tape data sets with standard labels are to be processed in a PL/I object program in which no non-sequential DASD data sets are to be processed.

If the LBLTYP statement is necessary as defined above, its omission will cause errors during execution of the program produced by the linkage editor. The DOS label-processing routines are loaded at execution time into storage reserved for them by the linkage editor if the LBLTYP statement is present. If this storage area is not available, the label-processing routines will overwrite a portion of storage occupied by the link-edited program.

The LBLTYP statement must precede the EXEC LNKEDT statement. (One LBLTYP statement only should be given.)

Note: Under DOS/VSE with Advanced Functions, label-processing storage is dynamically allocated, and the LBLTYP statement is not required.

OPTION Statement

The following options of the OPTION job control statement apply to the use of the linkage editor program:

```
LINK

CATAL
```

For a job that involves a link-editing step, the LINK option of the OPTION statement must be specified. If the output of the linkage editor is to be cataloged into the core-image library after the link-editing step, the CATAL option of the OPTION statement must be specified. When CATAL is specified, there is no need to specify LINK.

The OPTION statement must precede the EXEC LNKEDT statement for the linkage editor job step. Only one OPTION statement is needed for a job that contains more than one linkage editor step.

LINKAGE EDITOR CONTROL STATEMENTS

Linkage editor control statements are used to control the operation of the linkage editor. The statements are:

ACTION:	To specify linkage editor options.
INCLUDE:	To specify inclusion of additional relocatable object modules.
PHASE:	To name the link-edited phase.
ENTRY:	To define the initial entry point of the phase.

This section describes each linkage editor control statement as it applies to PL/I users' requirements.

Linkage editor control statements may be read from SYSRDR by job control, which copies the statements and any physically accompanying object module (on punched cards, for instance) onto the data set (SYSLNK) that is to contain the input to the linkage editor. Hence, all such statements in the input stream must precede the EXEC statement which invokes the linkage editor.

FORMAT OF CONTROL STATEMENTS: The first character of a linkage editor control statement is always blank. The control statement keyword (ACTION, INCLUDE, PHASE,or ENTRY) follows to the right of the first blank and may be preceded by further blanks. Operands must be separated from the statement keyword by one or more blanks and from each other by commas. Operands cannot extend past column 71. An operand field is delimited by a blank.

ACTION Statement

The ACTION statement is used to specify linkage editor options. When used, it should appear in the SYSRDR or SYSIN data set immediately after the OPTION LINK or OPTION CATAL statement so that it is the first record to be written onto SYSLNK. The options of the ACTION statement are:

```
CANCEL
F1 or F2
MAP or NOMAP
NOAUTO
```

Options can be given in any order.

CANCEL OPTION: The CANCEL option causes immediate cancellation of the job after the link-editing step if any errors occur that cause linkage editor diagnostic messages to be printed. These messages (numbered from 2100I to 2170I) are documented in the DOS System Control and System Services publication.

If this option is not specified, the linkage editor will list the errors and attempt to continue.

F1 AND F2 OPTIONS: The options F1 and F2 specify the foreground partition in which the link-edited program is to be executed. The use of either option requires that the operating system has been generated for multiprogramming, and also that each partition is allocated the same storage area at link-edit that will be allocated when the object program is executed. If used in a non-multiprogramming environment, either option is ignored. Programs with a phase origin (see "PHASE Statement" on page 50) specified as 'S', '*', or 'ROOT' can be designated for execution in a foreground partition by means of these options.

MAP OR NOMAP OPTION: The MAP option causes the linkage editor to produce a listing that indicates the main storage addresses assigned to each program phase and entry point processed in the job step. An example of this listing is given in Appendix A on page 281. The NOMAP option suppresses the listing. The option MAP is assumed if neither option is specified and SYSLST is assigned at link-edit time.

NOAUTO OPTION: The NOAUTO option causes the suppression of the AUTOLINK feature. AUTOLINK causes the automatic search of the relocatable library for resident library modules and any other object modules that are to be incorporated into the executable object program. The search takes place when an external reference cannot be resolved within the module itself. The AUTOLINK feature is recommended for use with all single-phase PL/I programs.

If no ACTION statement is given and SYSLST is assigned, ACTION MAP is assumed; if SYSLST is not assigned, ACTION NOMAP is assumed.

An example of an ACTION statement that requests the linkage editor to construct an executable program phase for execution in the currently-assigned foreground-1 partition follows:

```
ACTION F1
```

Additional features of the ACTION statement under DOS/VS include F3 and F4 for extra partitions, and REL and NOREL for the relocating loader.

The relocating loader can load any relocatable phase into any problem program partition. If ACTION REL is specified, a phase will be marked relocatable if possible, depending on the PHASE statement (see "PHASE Statement," below). ACTION REL is the default for systems that have relocating loader support specified at system generation time. DOS/VSE with Advanced Functions makes all programs relocatable unless NOREL is specified.

INCLUDE Statement

The INCLUDE statement can be used to specify that a module from the relocatable library is to be included in the link-editing operation, and is to be link-edited to form either a part of a phase or a complete phase. One or more relocatable object modules can be included in any one phase. The position of the INCLUDE statement determines the position of the included module within the phase.

The format of the INCLUDE statement is:

```
INCLUDE [modulename]
```

"modulename" specifies the name of the module to be included from the relocatable library. The module name must be the one specified in the optimizing compiler CATALOG option or the DOS Librarian CATALR statement when the module was cataloged. (See Chapter 6.)

When the INCLUDE statement is used without the "modulename" operand, the object module is assumed to follow the INCLUDE statement immediately in the output to the SYSIPT device. This method is normally used when the relocatable module is in the form of a deck of punched cards. (You can obtain an object module as punched cards by specifying the compiler DECK option when the program is compiled.)

INCLUDE statements may also be used to nest modules that are in the relocatable library. For example, a module included by use of an INCLUDE statement may itself cause another module to be included. Such nesting is permitted up to a depth of six levels. For example, a module included by an INCLUDE statement read from the card reader (SYSRDR) is at the <u>first level</u> and a module included for a first level module is at the <u>second level</u>.

An example of an INCLUDE statement that requests the linkage editor to incorporate an object module from a relocatable library into the executable program phase follows:

```
INCLUDE ZRE001
```

PHASE Statement

The PHASE statement specifies a name and the origin point of a phase to be produced by the linkage editor. The name is that with which the phase is cataloged in the core-image library, and by which it is identified for subsequent retrieval. The origin point specifies the location in main storage at which the phase is to be loaded for execution.

The compiler option NAME, if used, will cause the compiler to generate a PHASE statement preceding the object module produced for the compilation on SYSLNK. The NAME option is described in Chapter 4 of this publication.

The format of the PHASE statement is as follows:

```
PHASE name,origin[,NOAUTO]
```

where "name" is the name to be used for cataloging the phase in the core image library; "origin" is the symbolic or absolute location in main storage at which the phase is to be loaded; and NOAUTO optionally suppresses use of the automatic library (AUTOLINK) feature for the phase.

NAME OPERAND OF THE PHASE STATEMENT: The name operand represents the symbolic name of the phase. It may consist of one to eight alphameric characters. In a multiphase program, the phase name must be at least five characters, the first four of which are identical to those in the names of the other phases in the program but different from any other phase names in the library that are not part of the program.

An asterisk is not permitted as the first character of a phase name.

ORIGIN OPERAND OF THE PHASE STATEMENT: The origin operand of the PHASE statement specifies the load address in main storage to be used for the phase. The load address can be specified symbolically in one of the following formats:

1. symbol
2. *
3. ROOT
4. S
5. F + address

A complete list of the available forms is described in detail in <u>DOS/VSE System Control Statements</u>. Those given above are the most likely to be used when link-editing PL/I programs.

The first form, "symbol", specifies that the origin is to be identical to that of a phase that has already been processed by the linkage editor. Therefore, the symbol given should be the phase name used for a phase that precedes it in the input to the linkage editor. In effect, the phase currently processed will occupy the same storage as the phase named as the symbol. This mechanism is of use in constructing an overlay (multiphase) program.

The second form, "*", specifies that the origin is to be the next available doubleword address after the previous phase, if any. If there is no previous phase, the address is assumed to be the first doubleword address after the supervisor and any storage reserved for label processing, unless the ACTION statement is specified with the foreground option. In this case the phase origin will be at the start of either the foreground-1, foreground-2, or any partition specified, plus that of any preceding phase.

The third form, "ROOT", specifies that the phase is the root phase of an overlay (multiphase) program. This phase should be the first to be processed by the linkage editor. Its origin will be the next available doubleword address after the supervisor and any storage reserved for label processing, unless the ACTION statement is specified with the F1 or F2 option. In this case the phase origin will be at the start of either the foreground-1 or the foreground-2 partition.

The fourth form, "S", specifies that the phase origin is the next doubleword address available after the supervisor and any storage reserved for label processing, unless the ACTION statement is specified with the F1 or F2 option. If this is the case, the phase origin will be at the start of either the foreground-1 or foreground-2 partition.

The fifth form, "F + address", permits the specification of a foreground-1 or foreground-2 partition address as the origin of the phase. This form can be used instead of the ACTION statement with the F1 or F2 option and the PHASE statement with the "*", "S", or "ROOT" forms if the foreground partitions allocated when the link editing is performed do not occupy the same areas of main storage as the foreground partitions used when the program is executed. The "address" is an absolute storage location to which is added storage for a register save area and, if required, storage for label processing.

The first four forms are compatible with the generation of a relocatable phase unless they refer to another phase that is not relocatable. The last form will generate a nonrelocatable phase.

Refer to DOS/VSE System Control Program for more detail on the PHASE statement.

A PHASE statement is not required for a single-phase program that is to be written into the temporary area of a core-image library. The PHASE statement is required for a temporary multiphase program, however. (In either case, the EXEC statement that invokes a program phase from the temporary area does not specify a program name.)

A PHASE statement must precede the first object module in the phase to be produced by the linkage editor. Any object module not preceded by a PHASE statement will be included in the current phase.

When several PHASE statements occur, each of the PHASE statements must be followed by at least one INCLUDE statement to obtain a module from the relocatable library or the input stream. For example:

```
// JOB EXAMPLE
// OPTION CATAL
   PHASE PHAS1,ROOT
   INCLUDE PROGA
   INCLUDE PROGB
   PHASE PHAS3,*
   INCLUDE
         .
         .
        object module C
         .
         .
/*
// EXEC LNKEDT
/&
```

See also "Overlay (Multiphase) Programs" on page 53.

ENTRY Statement

The ENTRY statement can be used to specify the initial entry point in a phase to be invoked by the control program. The format of the ENTRY statement is as follows:

```
ENTRY entry-point
```

An ENTRY statement is not required unless the control section that receives control initially from the operating system is not the first control section to be processed by the linkage editor. For PL/I programs, the first control section in an object module is the compiler-generated control section PLISTART. This control section contains the entry-point address to the PL/I program.

An example of an ENTRY statement that causes control to be passed to an entry point other than that of the first control section in the link-edited phase follows:

```
ENTRY EPZ001
```

Example of Control Statements

The following gives the linkage editor INCLUDE and PHASE statements and the job control statements associated with the use of the linkage editor to create a multiphase program consisting of modules from a relocatable library.

```
// JOB FIG0500X
// OPTION CATAL
  PHASE PHAS1,ROOT
  INCLUDE A
  PHASE PHAS2,*
  INCLUDE B1
  PHASE PHAS3,PHAS2
  INCLUDE B2
// EXEC LNKEDT
/&
```

LINKAGE EDITOR LISTING

The listing produced by the linkage editor contains a statement of the options given in any ACTION statement or assumed by default, a listing of the INCLUDE, PHASE, and ENTRY statements processed, and a listing of the external references resolved by the AUTOLINK feature.

If the option MAP is given, the linkage editor also produces a listing showing the types of phase produced (ROOT or OVERLAY), the load-point address (XFR-AD), the phase origin address (LOCORE), the highest main storage address used (HICORE), the core-image library disk storage address used for the phase, and the relocation factor used for the phase. The contents of the phase and the control sections that have been combined by the linkage editor into the phase are then listed with their main storage start-addresses and the relocation factor used to obtain the main storage start-address. For each control section, each external entry point is listed with its main storage address. Unresolved external entry points are flagged with an asterisk. For every unresolved weak EXTRN, there is an unresolved message indicating an address constant.

Any errors encountered by the linkage editor are indicated by diagnostic messages with message numbers prefixed by "21."

An example of a linkage editor listing is given in Appendix A on page 281.

OVERLAY (MULTIPHASE) PROGRAMS

Programs larger than the available main storage space can often be subdivided into smaller phases that can fit into the available space and be executed independently of other phases in the program. During execution of such a multiphase program phases can be loaded and executed successfully in the same area of main storage. One phase must remain in main storage throughout; this is termed the root phase. The other phases are termed overlay phases. Both a root phase and an overlay phase can contain one or more PL/I external procedures, but the main procedure must always be in the root phase.

Control cannot be passed from a PL/I procedure in the root phase to a procedure in an overlay phase until the overlay phase has been loaded into main storage. Loading is achieved by means of the resident library module PLIOVLY. The module PLIOVLY is invoked by the statement:

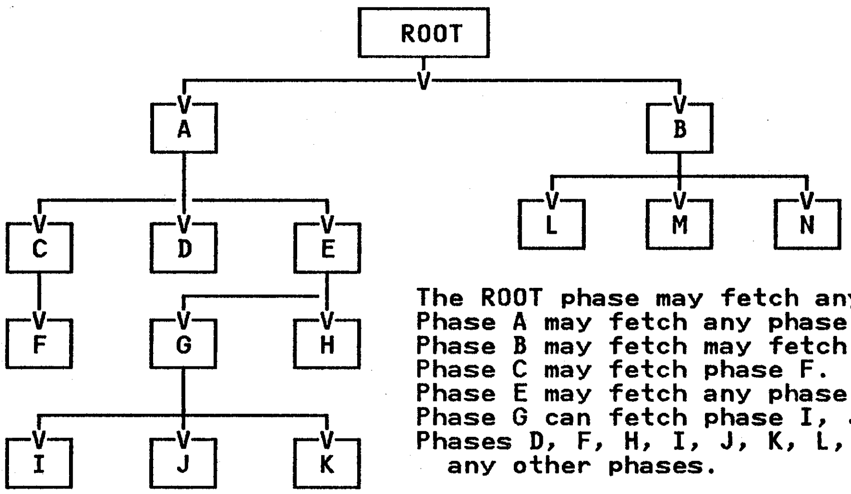
```
CALL PLIOVLY (character-string-expression);
```

where "character-string-expression" represents the name of an overlay phase in the core-image library. The phase name used in a CALL PLIOVLY statement is a unique name given in the NAME option or PHASE statement when the overlay phase was created. For example:

```
CALL PLIOVLY ('PHASE5');
```

The phase is then loaded into main storage and control is returned to the PL/I procedure. The statement immediately following the CALL PLIOVLY statement is then executed. Entry points within the overlay phase can then be invoked by a CALL statement or a function reference from any phase currently resident in main storage at any time after the containing phase has been loaded, provided that it has not been overwritten by a subsequently loaded phase.

Note that an overlay phase can itself initiate the loading of further phases. Nesting of overlay phases can take place up to a depth of six. However, the calling phase must not itself be overwritten in main storage by the newly-loaded overlay phase. Figure 12 shows an overlay program as a tree structure. The use of a tree structure can help in planning an overlay program.



The ROOT phase may fetch any phase A through N.
Phase A may fetch any phase C through K.
Phase B may fetch may fetch any phase L through N.
Phase C may fetch phase F.
Phase E may fetch any phase G through K.
Phase G can fetch phase I, J, and K.
Phases D, F, H, I, J, K, L, M and N cannot fetch any other phases.

Figure 12. Overlay Structure

An overlay program phase can fetch into main storage any overlay phase that is directly below it in its tree structure hierarchy. If an overlay phase is more than one level below its fetching phase, space is reserved for any intermediate phase that may subsequently be fetched into storage.

Constructing Overlay Programs

An overlay program is constructed by the DOS linkage editor from a number of relocatable object modules. The linkage editor can obtain these modules from a preceding batched compilation, or from a relocatable library, or from a device assigned to SYSIPT.

Each group of relocatable object modules that is to form a single phase in an overlay program must be preceded in the input to the linkage editor by a PHASE statement. The format of the phase statement is described earlier in this chapter.

The root phase of a PL/I overlay program must contain at least one PL/I procedure with the MAIN option. The programmer must ensure that the root phase is not overlaid by any other phases during execution.

Note: If you want to link-edit the output from a PL/I batched compilation in the immediately following job step, use the compiler NAME option to generate the appropriate PHASE statement for each overlay phase.

PL/I Resident Library Modules

It is probable that most of the PL/I object modules in an overlay program will contain references to modules of the DOS PL/I Resident Library. The location of these modules within the overlay structure can significantly affect the space requirements of the program. You must therefore decide in which phase or phases of your overlay program each required library module is to be placed.

The following sections give suggested techniques for optimizing the placement of resident library modules within the phases of an overlay program. <u>These techniques apply only to Release 28.0 of the Disk Operating System and to any subsequent releases</u>. If the release level of your operating system is earlier than 28.0, refer to "DOS Releases Before Release 28.0" on page 57.

Link-Editing Wide Overlay Structures

A <u>wide</u> overlay structure is one that contains few levels of overlay, but which contains many phases at each level. For a structure of this type, optimum use of available storage can usually be achieved by allowing each phase to contain the library modules that it calls, with the exception of certain modules that must always be in the root phase.

The required placement of library modules is achieved as follows:

1. Use the AUTOLINK feature of the DOS linkage editor; that is, do not code the NOAUTO option on any PHASE statement.

2. Include the following modules in the root phase if the associated function is used anywhere in the overlay program.

IBMBJWTA	if the wait statement is used.
IBMBTOCA	if the COMPLETION pseudovariable is used.
IBMBJDSA	if the DISPLAY statement with the EVENT option is used.

IBMBRIOA if the EVENT option is used on a record I/O statement.

IBMBSCPA if the GET statement with COPY option is specified.

3. Include module IBMDPOLA in every overlay phase (but not in the root phase).

Note: Because the position of an INCLUDE statement determines the position of the module within the phase, the INCLUDE statements for the resident library modules must follow the PL/I source code.

When the overlay program is processed by the linkage editor, the AUTOLINK feature ensures that external references to modules whose names begin with "IBM" are resolved within the phase in which they occur. Each phase thus gets those library modules that it requires and no others. The INCLUDE statements for the root phase ensure that the modules listed in (2) above are placed in the root phase. Module IBMDPOLA contains dummy entry points for modules that must be in the root phase, and prevents additional copies of these modules being placed in the overlay phases.

If any overlay phase contains a CHECK condition prefix, ensure that a CHECK condition prefix for a variable with the same attributes occurs in the root phase. It may be necessary to insert a dummy variable in the root phase to satisfy this requirement.

An example of the link-edit procedure for a wide overlay structure follows:

```
// JOB
// OPTION LINK
   PHASE PHASER,ROOT
   .
   (Main procedure using PLISORT
   built-in function)
   .
   INCLUDE IBMBPGRA
   PHASE PHASE1,*
   (phase 1 object module)
   INCLUDE IBMDPOLA
   PHASE PHASE2,PHASE1
   (Phase 2 object module)
   INCLUDE IBMDPOLA
/*
//
/&
```

Link-Editing Tall Overlay Structures

A <u>tall</u> overlay structure is one that contains many levels of overlay, but few phases at each level. For a structure of this type, optimum use of available storage can usually be achieved by including commonly used library modules in the root phase. This can be done as follows:

1. Code the NOAUTO option on each PHASE statement except the PHASE statement for the root.

2. Create a list of library modules that are to be included in the root phase. The list can be created in one of two ways:

    a. Examine all external procedures in the compiler ESD listings for all the overlay phases of the program, and select all module names that are marked "ER."

    b. Perform a trial link-edit on the program, and select all module names that are marked as unresolved external references (EXTRN) in the linkage editor listing.

3. Place INCLUDE statements for the selected modules in the root phase. An example of the link-edit procedure for a tall overlay structure is shown in Figure 13.

<u>Stage 1 - Trial Link Edit</u>

```
// JOB
// OPTION LINK
   PHASE   PHASER,ROOT
   .
   (Procedure with references to
    library modules "A" and "B")
   .
   PHASE    PHASE1,*,NOAUTO
   .
   PHASE    PHASE2,PHASE1,NOAUTO
   .
   (Procedure with references to
    library modules "B," "C," and "D")
    .
/*
//  EXEC   LNKEDT
/&
```

The output from the linkage editor shows that references to library modules "C" and "D" are unresolved. INCLUDE statements for these modules must therefore be added to the root phase.

<u>Stage 2 - 2nd Link Edit</u>

```
// JOB
// OPTION LINK
   PHASE    PHASER,ROOT
   INCLUDE C
   INCLUDE D
   .
   .
   PHASE    PHASE1,*,NOAUTO
   .
   .
   PHASE    PHASE2,PHASE1,NOAUTO
   .
   .
/*
// EXEC  INKEDT
/&
```

Figure 13. Link-Editing a Tall Overlay Structure

Link-Editing Complicated Overlay Structures

An overlay program may be both wide and tall, or it may contain some wide branches and some tall branches. In these cases, the link-editing techniques described previously may result in a program size that is considerably larger than optimum, either because the root phase contains many library modules that are used very infrequently, or because many copies of the same module are in storage at the same time.

The following sections describe techniques that can be used to improve the overlay program. It is assumed that an initial link-edit has been performed on the basis of whether wide or tall structures predominate in the program.

Improvements to Overlay Programs Link-Edited with AUTOLINK

If an overlay program has been link-edited using the AUTOLINK feature, the storage requirements of tall branches in the program may be unnecessarily large because many library modules occur in more than one phase in the branch. If this is the case, the situation can be improved by moving library modules to the highest common phase in the branch. This can be done as follows:

1. Add the NOAUTO option to the PHASE statements for all phases in the branch except the first.

2. Remove the INCLUDE IBMDPOLA statement from all phases in the branch except the first.

3. Working from the ESD listings of the compilations for the changed phases, create a list of library modules by selecting each module name that is marked "ER." References to modules IBMBJWTA, IBMBTOCA, IBMBPIRA, IBMBPGRA, and IBMBERCA should be ignored, because IBMDPOLA contains dummy entry points for these modules.

4. Insert INCLUDE statements for the selected modules in the first phase of the branch.

Improvements to Overlay Program Link-Editing with NOAUTO

If an overlay program has been link-edited using the NOAUTO option, the root phase may be unnecessarily large because it contains library modules that are used only by a few phases in the program. In this case, the situation can often be improved by moving these library modules from the root phase to the phases that require them.

Only modules that do not call other library modules can be treated in this way. A list of library modules that do not call other library modules, and which therefore can be placed anywhere in an overlay program, is given in Figure 14 on page 58.

Modules can be moved from the root phase as follows:

1. Working from the compiler ESD listings, select modules that are listed in Figure 14 on page 58 and which are used infrequently.

2. Place an INCLUDE statement for each selected module in each phase that uses it, and remove the INCLUDE statements for these modules from the root phase.

DOS RELEASES BEFORE RELEASE 28.0

If your disk operating system is earlier than release 28.0 the DOS linkage editor will not recognize that special treatment of PL/I resident library modules is required, and will treat them just like any other external module. That is, the linkage editor will include each library module once only, in the first phase in which it is requested.

Thus it is possible that a phase containing a particular module will not have been loaded or will not have been overlaid by the time another phase that requires it is executed.

To guard against this possibility, use the techniques described under "Link-Editing Tall Overlay Structures" earlier in this chapter.

DATA VARIABLES AND FILES IN OVERLAY PROGRAMS

Variables with the attributes STATIC INTERNAL or AUTOMATIC in an overlay phase will have undefined values if the phase is overlaid and a new copy of the same phase is subsequently loaded and invoked. Consequently, any variables with values that are

IBMBAMMA	Structure mapping
IBMBAYFA	POLY built-in function
IBMBBBAA	AND, OR operations (byte-aligned bit strings)
IBMBBBNA	NOT operation (byte-aligned bit strings)
IBMBBCIA	INDEX (character strings)
IBMBBCKA	Concatenate, REPEAT (character strings)
IBMBBCTA	TRANSLATE (character strings)
IBMBBCVA	VERIFY (character strings)
IBMBBGCA	Compare (general bit strings)
IBMBBGFA	Assign (byte-aligned bit strings) and Fill (general bit strings)
IBMBBGIA	INDEX (bit strings)
IBMBBGSA	SUBSTR built-in function
IBMBBGVA	VERIFY (bit strings)
IBMBCBBA	Conversion (bit to bit)
IBMBCBCA	Conversion (bit to character)
IBMBCBQA	Conversion (bit to pictured character)
IBMBCOA	Conversion (packed decimal to pictured decimal)
IBMBCVA	Conversion (packed decimal to E format)
IBMBCWA	Conversion (packed decimal to F format)
IBMBCYA	Conversion (fixed binary to fixed binary and float to float)
IBMBEOCA	ON-code
IBMBEOLA	ONLOC built-in function
IBMBJDTA	DATE built-in function
IBMBJDYA	DELAY
IBMBJDZA	DISPLAY without EVENT
IBMJTTA	TIME built-in function
IBMBKCPA	Checkpoint/restart interface
IBMBKSTA	SORT interface
IBMBMPUA	MULTIPLY (fixed binary complex)
IBMBMQUA	DIVIDE (fixed binary complex)
IBMBMRUA	ABS (fixed binary complex)
IBMBMRVA	ABS (fixed decimal complex)
IBMBMUDA	Shift and assign/load (fixed decimal real)
IBMBMVUA	Multiplication and Division (fixed binary complex)
IBMBMVVA	Multiplication and Division (fixed decimal complex)
IBMBMVWA	Multiplication (long and short float complex)
IBMBMWXA	Division (short float complex)
IBMBMWYA	Division (long float complex)
IBMBMXLA	Integer exponentiation (long float real)
IBMBMXSA	Integer exponentiation (short float real)
IBMBPAFA	Controlled variable management
IBMBPAMA	AREA management
IBMBPRCA	Return code module

Figure 14. Library Modules that Can Be Placed Anywhere in an Overlay Program

to be used by a subsequent copy of the same phase should be declared with the attributes STATIC EXTERNAL both in the overlay phase and in the root phase.

Variables with the attributes CONTROLLED EXTERNAL in overlay phases must also be declared in the root phase of the program.

Variables whose values are used in more than one overlay phase should also be declared STATIC EXTERNAL in each overlay phase that uses them and in the root phase. This will cause all such variables to be stored in a common area at the beginning of the problem program partition.

If STATIC EXTERNAL data is shared by two or more phases at two or more levels below the root phase, it is possible to store such data in a common phase at a higher level rather than in the root phase.

This is done by declaring the STATIC EXTERNAL data in the common phase rather than in the root phase. The declaration must also include the INITIAL attribute, to force the data item to be mapped in the storage area of the common phase.

For example, in Figure 12 on page 53, suppose that phases C and D use a large array that is not used in phase B or any of its subsidiary phases. The storage for such an array is allocated throughout execution of the program if it is declared STATIC EXTERNAL in the root phase. If it is declared STATIC EXTERNAL INITIAL (...) in phase A, it will be allocated storage only when A is loaded and the PL/I procedure within A is invoked. The storage will be allocated within the storage area for this procedure. Thus the storage available to phase B and to any of its subsidiary phases is increased by the amount of storage the array would otherwise have occupied.

Note that the values in the array will be lost as soon as phase A is overlaid by phase B. Consequently, this technique must not be used when the values of the STATIC EXTERNAL data are to be preserved throughout any overlaying and reloading of the common phase.

Variables whose values are used in more than one phase can also be passed as arguments in CALL statements or function references. However, the values of such variables will be undefined when the invoking procedure is overlaid, unless they are STATIC EXTERNAL in both the overlay phase and the root phase.

If a file is declared and opened in an overlay phase, the file must be closed before the phase is overwritten. However, the file need not be closed if it was also declared in the root phase. As the SYSPRINT file is used for error messages, the CSECT for the file should be included in the root phase; that is, it should be declared explicitly or implicitly in the root phase.

OVERLAY PROGRAMS IN A FOREGROUND PARTITION

If an overlay program is to be link-edited for execution in a foreground partition, the ACTION statement must be used to specify the required foreground partition, since the PHASE statement for the root phase must specify "ROOT," and therefore cannot be used to specify a foreground origin address. Note that the foreground partition should be allocated exactly as it will be allocated when the overlay program is executed. If the first three characters of the phase names in a foreground overlay program are 'FGP', the phase will be retrieved and loaded from the core-image library more efficiently.

Examples of Overlay Program Creation

The following two examples show the use of the optimizing compiler and the linkage editor in creating an overlay program.

The example of Figure 15 on page 60 shows several PL/I object modules and other relocatable modules from both the relocatable library and the input stream that are written onto the SYSLNK data set prior to the link-editing step that creates a multiphase program. However, if the first module to be invoked cannot be the first module in the input, it must, with some exceptions, be supported by a linkage editor ENTRY statement to identify it as the first module to be invoked.

In Figure 15, an overlay program is link-edited, written onto the core-image library, and executed. The root phase of the program is a phase called OVLAY1. It consists of a main PL/I procedure compiled in the first job-step, and an object module identified in the relocatable library as RELMOD1.

The overlay program is to be executed in the background partition. It will occupy storage immediately following the resident supervisor program and a communications area.

```
// JOB OVERLAY1
// OPTION CATAL
 PHASE OVLAY1,ROOT
// EXEC PLIOPT,SIZE=64K
  RT:PROCEDURE OPTIONS(MAIN);
      DECLARE (X,E) ENTRY;
  CALL PLIOVLY ('OVLAY2')
      .
      .
  Y = X * Z; /* FUNCTION X INVOKED*/
      .
      .
  CALL PLIOVLY ('OVLAY3');
      .
      .
  CALL E;
      .
      .
  END RT;
/*
 INCLUDE RELMOD1
 PHASE OVLAY2,*
 INCLUDE
      .
      .
      object module card deck
      (entry point "X")
      .
      .
/*
 PHASE OVLAY3,OVLAY2
 INCLUDE RELMOD2
 INCLUDE
      .
      .
      object module card deck
      (entry point "E")
      .
      .
/*
// EXEC LNKEDT
// EXEC OVLAY1
/&
```

Figure 15. An Overlay Program

The first overlay phase, OVLAY2, is to occupy the next available storage location after the root phase. This phase consists of a relocatable object module in the form of a card deck. The first overlay phase is invoked at entry point X. The second overlay phase (OVLAY3) is to occupy the same storage locations as OVLAY2. OVLAY3 consists of the object module RELMOD2 from the relocatable library and the second relocatable object program submitted as a card deck. The second overlay phase is invoked at entry point E. When procedure RT is executed, the overlays are loaded successively, the second overwriting the first.

The second example in Figure 16 on page 61, shows the use of the PL/I batched-compilation facility to compile several external procedures from one invocation of the compiler and to link-edit them to the background partition.

In this example, the PHASE statements are generated by the optimizing compiler. A PHASE statement is produced for every compilation for which the compiler option NAME is used in the PROCESS statement. The operands of the NAME option are specified exactly as for the PHASE statement. Each generated PHASE statement is placed before the object module produced by the compilation on SYSLNK.

```
// JOB FIG0503X
// OPTION CATAL
// EXEC PLIOPT
* PROCESS NAME('FGPOVL1,ROOT');
   RT:PROCEDURE OPTIONS(MAIN);
      DECLARE (E1,E2) ENTRY;
     .
     .
     CALL PLIOVLY ('FGPOVL2');
     .
     .
     CALL E1;
     .
     .
     CALL PLIOVLY ('FGPOVL3');
     .
     .
     CALL E2;
     .
     .
     END RT;
* PROCESS NAME('FGPOVL2,*');
   E1:PROCEDURE;
     .
     .
     END E1;
* PROCESS NAME('FGPOVL3,FGPOVL2');
   E2:PROCEDURE;
     .
     .
     END E2;
// EXEC LNKEDT
// EXEC FGPOVL1
/&
```

Figure 16. Overlay Program Using Batched Compilation

The CATAL option in the OPTION statement specifies that the linkage editor output is to be cataloged on the core image library.

FGPOVL1 is the root phase; FGPOVL2 and FGPOVL3 are overlay phases that occupy the same main storage locations. The layout of the main storage partition for this program is given below:

ROOT PHASE AREA - FGPOVL1
OVERLAY PHASE AREA - FGPOVL2 and FGPOVL3

Use of Overlay Phases

The following notes describe the possible organization of an application program into an overlay program.

Note that the figures used are not meant to be representative but merely to illustrate the points that are being explained.

Assume that the program is large, containing a total of approximately 2000 statements. It is to be executed in the background partition (BG), in which there are 48K bytes of main storage. (It is assumed for the purposes of this discussion that this partition is permanently allocated this amount of main storage so that the PL/I Optimizing Compiler can be used regularly.) If compiled as a single-phase program, the PL/I object program would require approximately 150K bytes of main storage for execution. The storage requirements for the program might consist of the following:

STATIC EXTERNAL data	5K
AUTOMATIC and CONTROLLED data	Up to 10K at any one time
Compiled object code (including link-edited PL/I library subroutines and space for transient library routines)	130K
Total	145K bytes

However, it is reasonable to expect that a program of this size would be developed and tested initially as a number of separate external procedures. Consequently, the creation of an overlay program with phases containing one or more external procedures can be envisaged at an early stage in the development of the program.

To enable this program to be executed in a 48K-byte-partition, it must be divided into a root phase and a number of overlay phases. The root phase is designed to load and invoke each overlay directly. The background partition will contain the root phase and one overlay phase only at any one time. The storage requirement for the root phase is:

STATIC EXTERNAL data	5K
AUTOMATIC and CONTROLLED data	1K
Compiled object code (excluding library subroutines)	10K
PL/I resident library subroutines (some of which are used by overlay phases)	8K
Space allowed for transient library routines	4K
Total	28K bytes

The approximate maximum storage requirements for an overlay phase is:

STATIC EXTERNAL data	None
AUTOMATIC and CONTROLLED data	Up to 10K at any one time
Compiled object code (excluding library subroutines)	6K
PL/I resident library subroutines (some of which are used exclusively by the overlay phase)	2K
Space allowed for transient library routines	2K
Total	20K bytes maximum

The combined storage for the root phase and the largest of the overlay phases should not exceed 48K bytes.

These figures are approximations only. The subdivision of a single program will probably cause a marginal increase in the total storage requirement and a significant increase in the total execution time because of increased overheads in phase loading and block initialization. The assumptions shown above are for guidance only. The requirements for a particular multiphase program can only be determined locally.

The size of a PL/I object module before link-editing can be obtained from the compiler listing entitled "Storage Requirements." The resident library modules that are required for the object module are given in the external symbol dictionary (ESD) listing, also produced by the compiler. Resident library modules for the optimizing compiler have the characters "IBMB" in the first four positions of the library module name. The total length of a link-edited program phase is given in the storage map produced by the linkage editor for each program phase.

LINKING PL/I AND OTHER LANGUAGE MODULES

Relocatable object modules written in various programming languages, such as PL/I, COBOL, FORTRAN, and assembler language, can be combined by the linkage editor to form a single executable program. This section describes some of the methods that can be used to achieve the correct combination of modules from PL/I and other languages.

The first consideration is to determine which module is to receive control initially when the program is invoked and to ensure that the linkage editor takes the appropriate action for this module. A second consideration is the use of the linkage editor itself, particularly when the object modules to be processed are to be obtained from different sources.

Establishing Initial Control

The main consideration in using the linkage editor to combine relocatable object modules, particularly if they were written in different programming languages, is to determine which module is to receive initial control. Normally, the first object module to be processed by the linkage editor will automatically be the first module in the executable program to be invoked. However, if the first module to be invoked cannot be the first module in the input, it must, with some exceptions, be supported by a linkage editor ENTRY statement to identify it as the first module to be invoked. Figure 17 on page 64 shows when an ENTRY statement should or should not be used.

Linking Multiple Object Modules

One method of linking multiple object modules into a single executable program is to compile (or assemble) each module and catalog the resulting object module in a relocatable library. The link-editing run would require the appropriate linkage editor INCLUDE statements in the order in which the modules are to be retrieved from the relocatable library and combined. If two modules contain the same entry point name (this occurs in some IBM-supplied modules when a superset module contains the same entry point name as a subset module), this method prevents a linkage editor message 2143I (duplicate entry point label).

Another method is to compile and link edit the modules as one job stream. If two modules contain the same entry point name, an INCLUDE statement for the superset module must be the first statement to the linkage editor.

A third method is to produce the object modules in the form of card decks for direct input to the linkage editor. It is also possible to perform a series of compilations and assemblies in the same job; this results in a sequence of object modules being written onto SYSLNK for direct input to the linkage editor in a link-editing step that immediately follows the last compilation or assembly in the job. This technique is used in an example in Chapter 11. It is a convenient method of performing multiple-compile, link, and execute operations within a job for checking out such programs.

Type of Module Containing Initial Entry Point	Type of Module First in Editor Input	Is ENTRY Required?
PL/I	COBOL Assembler with END label statement Assembler without END label statement FORTRAN[3]	YES[1] YES[1] NO NO
COBOL 'D'	PL/I Assembler with END label statement Assembler without END label statement FORTRAN	Not permitted, therefore the COBOL module <u>must</u> be the first in the input to the linkage editor
COBOL 'F'	PL/I Assembler with END label statement Assembler without END label statement FORTRAN	YES YES NO NO
FORTRAN	PL/I[3] COBOL Assembler with END label statement Assembler without END label statement	YES YES YES YES
Assembler - with END label statement.	PL/I COBOL FORTRAN	YES YES YES[2]
Assembler - without END label statement	PL/I COBOL FORTRAN	YES YES YES
RPG II	RPG II Any other	NO YES

[1]The entry name used in the ENTRY statement must be PLISTART.
[2]This applies when PL/I or COBOL modules are also present; if the program consists of assembler and FORTRAN modules only, the ENTRY statement is not required.
[3]If the PL/I and FORTRAN modules have common STATIC EXTERNAL storage defined by means of a FORTRAN COMMON statement, the PL/I object module <u>must</u> be the first module processed by the linkage editor.

Note:
A full list of COBOL and FORTRAN compilers can be found in the publications: <u>DOS PL/I Optimizing Compiler: General Information</u>, and <u>DOS PL/I Optimizing Compiler: Installation</u>.

Figure 17. Use of ENTRY Statement in Multilanguage Programs

Any combination of these techniques may be used to suit the requirements of a particular application.

If the program to be link-edited contains a mixture of PL/I and Assembler language modules in which the PL/I module is invoked from the Assembler language module, the resident library module IBMBPJRA must be included in the executable program. A linkage editor INCLUDE statement can be used for this purpose.

RELINK-EDITING FOR 3330-11 AND 3350 UNDER DOS/VS RELEASE 34

Programs running under DOS/VS Release 34 that want to take advantage of support for the 3330-11 and 3350 direct-access storage devices need not relink-edit their programs when all the following are true:

- The supervisor has rotational sensing (RPS) support generated in the system.

- RPS support has been loaded into the SVA, or sufficient SVA space is available for loading the RPS logic modules.

- The program is running in virtual mode and the SIZE operand is specified on the EXEC statement.

- The program has sufficient GETVIS space for RPS extensions to existing control blocks.

- The program currently specifies 2311, 2314, 3330, or 3340 DASDs for the files that are to be moved to the 3330-11 and 3350 DASDs.

  Note: Release 34 does not support 3370, 3375, or 3380 devices.

If these conditions are not true, relink-editing is necessary.

CHAPTER 6. PROGRAM LIBRARY CREATION AND MAINTENANCE

This chapter describes some of the basic features and requirements for cataloging and maintaining PL/I programs in the various types of libraries available under DOS/VSE with Advanced Functions. If you intend to make extensive use of the DOS/VSE Librarian facilities, particularly for creation and reorganization of the libraries described in this chapter, refer to VSE/Advanced Functions: System Control Statements.

PROGRAM LIBRARIES

Four types of program libraries can be used in DOS/VSE Advanced Functions:

- Core-Image Library—contains the executable programs, in core-image format, that are kept in a DOS/VSE installation.
- Relocatable Library—contains compiled, but not executable, object modules.
- Source Statement Library—contains programs in source statement form.
- Procedure Library—contains DOS/VSE job control statements.

These are system libraries, which are always available when a program is run.

In addition, private core-image, relocatable, and source statement libraries can be defined. A private library can only be established on a device similar to that used for the system library, a 2314, 3330, or other direct-access device. These private libraries are usually available when a program is run. They can be different for each partition in the system, or they can be the same for all partitions.

The number of private and system libraries, and the search order in which they are accessed, is determined by the DOS/VSE job control statements. When you search multiple libraries, the LIBDEF control statement can be used. To define multiple libraries, including the system libraries, use the SEARCH clause of the LIBDEF control statement. DOS/VSE uses the order in which the libraries are specified as the search order.

If only one set of private libraries is defined, an assignment of SYSCLB (private core image), SYSRLB (private relocatable), or SYSSLB (private source statement) can be used. In this case, the library defined by the assignment is used first to satisfy the request. If the defined file cannot be found in the private library, the system library is searched.

Only one library can be identified for output at any one time; you identify this library through the TO clause of the LIBDEF control statement.

Before issuing the LIBDEF or ASSIGN job control statement:

- The storage volume containing the library (or libraries) must be mounted.
- The symbolic device must be assigned a device address.
- DLBL control statements for each library to be accessed must be issued; these control statements make the library labels available to the system when the LIBDEF control statement is issued.

Each of these libraries is maintained by the DOS/VSE Librarian. The librarian programs can be used to insert, delete, and condense the contents of a library. Some of the uses of MAINT (one of the librarian service programs) are given in the following sections of this chapter.

CORE-IMAGE LIBRARY

A core-image library contains programs in the form of executable program phases. Any program to be executed in a DOS/VSE environment (that is, under control of the resident DOS/VSE supervisor program) must be stored in a core-image library before it can be loaded for execution. Programs can be stored either temporarily or permanently.

The temporary area of the private or system core-image library used as output can hold only one program; this temporary area is used for object programs that have been link-edited and are to be loaded in the next job step.

An executable program should not be permanently inserted in the core-image library until it is verified and complete.

Any number of private core-image libraries can be created. In a multiprogramming environment, a core-image library can hold:

- One copy of a non-self-relocating program for execution in only one partition.
- A relocatable program that can be executed in any partition.
- A self-relocating program that can be loaded from the core-image library for execution at any storage address.

The PL/I Optimizing Compiler, and any of the object modules it produces, is relocatable. A single copy can be link-edited into the system core image library and executed in any partition that is large enough for the program.

Including New Programs

The inclusion of a new executable program phase in the core-image library is performed after successful completion of the link-editing step for the program.

To catalog a program permanently in the core-image library:

1. Include the following job control statement in the job stream:

   ```
   // OPTION CATAL
   ```

2. Name the program and specify its loading address, either in the compiler NAME option, or in a linkage editor PHASE statement.

For a program phase that is required only for the duration of the job, include the following statement:

```
// OPTION LINK
```

Deleting Unwanted Programs

To delete one or more unwanted program phases from the core-image library, use the librarian program MAINT with a DELETC statement to specify the phase names of the programs to be deleted. The following example deletes two program phases (PROG1 and PROG2) from the core-image library:

```
// JOB CLEAR
// EXEC MAINT
 DELETC PROG1,PROG2
/*
/&
```

If the program to be deleted consists of a number of phases, an alternate form of the DELETC statement, requiring only one operand for the entire set of phases, is available. The following example deletes all the phases belonging to the same program, provided the first four characters of the phase names are identical (PROG). (The DELETC statement specifies these identical characters only.)

```
// JOB CLROVLY
// EXEC MAINT
 DELETC PROG.ALL
/*
/&
```

SOURCE STATEMENT LIBRARY

A source statement library contains any number of books, each consisting of a sequence of source statements. Any number of books can be included in a PL/I compilation by using the preprocessor facilities of the compiler and the %INCLUDE statement.

Books of assembler language, COBOL, and PL/I source statements are grouped into separate sublibraries. The sublibrary for PL/I source statements is designated P. For example, the following %INCLUDE statement requests the inclusion of the PL/I source statement book DCLSTMTA:

```
% INCLUDE P(DCLSTMTA);
```

Inserting a Source Statement Book

To include a sequence of PL/I source statements as a new book in the PL/I sublibrary of a system or private source statement library, use the librarian program MAINT and supply a CATALS statement that gives the sublibrary and book name of the new PL/I source statement book. The source statement records must be preceded and followed by a librarian BKEND statement. They must not include a PL/I PROCESS statement; the compilation will be in error if one is included. The following example shows a job in which a sequence of PL/I source records is cataloged in the system source library:

```
// JOB SOURCE
// EXEC MAINT
 CATALS P.DCLIB
 BKEND
 /* STANDARD DECLARATION OF DCLIB */
   DCL 1 RECQ,
         2 (RECA,
            RECB,
            RECC,
            RECD) CHAR(50),
         2 KEY,
           3 (FIRST,
              SECOND,
              THIRD) CHAR(3),
           2 CODE CHAR(1);
 /* END OF DCLIB */
 BKEND
/*
/&
```

Deleting Unwanted Source Statement Books

To delete a source statement book, use the librarian program MAINT with the DELETS statement to specify the name of the source statement book that is to be deleted. The following example deletes the book DCLIB created above:

```
// JOB DELSOURCE
// EXEC MAINT
 DELETS P.DCLIB
/*
/&
```

The space used for the source statement book DCLIB cannot be reused until the library is condensed.

RELOCATABLE LIBRARY

A relocatable library contains any number of relocatable object modules. A relocatable object module is a program that has been either compiled or assembled, but that cannot be executed until it has been processed by the linkage editor to form an executable program phase.

The term relocatable is used for an object module that can be link-edited for subsequent execution at any predefined storage location. Do not confuse relocatable with self-relocating, which is used for a particular type of executable program phase that can be loaded from the core-image library for execution in different storage locations on different occasions.

An object module produced by the optimizing compiler is in relocatable format, and may be included in a relocatable library.

Inserting a Relocatable Object Module

To add a relocatable object module to a relocatable library, use the librarian program MAINT with the CATALR statement. The CATALR statement specifies that the relocatable object module on the device assigned to SYSIPT is to be incorporated into the library and cataloged with the name given in this statement. The following example catalogs a relocatable object module (named MODULE1) supplied from a card deck in the card reader normally assigned to SYSIPT:

```
// JOB CATREL
// EXEC MAINT
 CATALR MODULE1
 .
 .    PL/I relocatable
 .    object module
/*
/&
```

A PL/I object module can be obtained as a deck of punched cards by specifying the DECK option for the compilation.

Compiling and Cataloging into a Relocatable Library

Two methods of cataloging a relocatable object module are available that do not require the object module in the form of a deck of punched cards. Both methods involve assigning a magnetic tape or disk storage device to SYSPCH for the compilation step, and reassigning the device to SYSIPT for a subsequent cataloging step. One method is provided by the operating system, the other by the optimizing compiler.

The DOS method recognizes a CATALR statement in the input stream preceding the EXEC statement for the compilation step, and copies the statement onto SYSPCH. In the subsequent compilation, the DECK option must be specified so that the object module created is written onto SYSPCH following the CATALR statement. This method is satisfactory for single compilations. The following is an example of cataloging an object module:

```
// JOB FIG0601X
// ASSGN SYSPCH,X'183',X'C8'
// MTC REW,SYSPCH
 CATALR MODULE1
// EXEC PLIOPT,SIZE=64K
* PROCESS DECK,NOLINK;
          .
          .
          .
/*
// MTC WTM,SYSPCH
// MTC REW,SYSPCH
// RESET SYSPCH
// ASSGN SYSIPT,X'183',X'C8'
// EXEC MAINT
/&
```

In this example, the MTC statements are used to write the end-of-file marker and rewind the magnetic tape volume. The symbolic device name SYSPCH must be reassigned its former device address before SYSIPT can be assigned the device address of the magnetic tape unit.

It is possible to perform subsequent link-editing and execution steps by omitting the NOLINK compiler option and providing the appropriate job control statements following an ASSGN statement that restores SYSIPT to its former device address.

The method provided by the optimizing compiler requires the use of both the DECK option and the CATALOG option. The compiler CATALOG option causes the compiler to generate a CATALR statement, using the name given with the option for the name of the relocatable object module that is to be cataloged. The CATALR statement precedes the object module that is written onto SYSPCH.

The compiler CATALOG option permits the use of batched compilation, in which more than one object module can be cataloged in a single step following the compilation step. This is illustrated in Figure 18 on page 71.

Deleting Unwanted Relocatable Object Modules

Use the librarian program MAINT with the DELETR statement to delete an unwanted object module from a relocatable library. The following example deletes the object module MODULE1:

```
// JOB DELETREL
// EXEC MAINT
 DELETR MODULE1
/*
/&
```

The space used for MODULE1 cannot be reused until the library is condensed.

```
// JOB FIG0601X
// ASSGN SYSPCH,X'183',X'C8'
// MTC REW,SYSPCH
// EXEC PLIOPT,SIZE=64K
* PROCESS CATALOG('MODULE1'),DECK,NOLINK;
     .
     .
     .
* PROCESS CATALOG('MODULE2'),DECK,NOLINK;
     .
     .
     .
* PROCESS CATALOG('MODULE3'),DECK,NOLINK;
     .
     .
     .
/*
// MTC WTM,SYSPCH
// MTC REW,SYSPCH
// RESET SYSPCH
// ASSGN SYSIPT,X'183',X'C8'
// EXEC MAINT
/&
```

Notes:

1. If a disk storage device is to be used for SYSPCH, and subsequently SYSIPT, both the compilation and the cataloging steps must be supplied with the appropriate DLBL and EXTENT statements.

2. If SYSPCH has a permanent (system-generated) assignment, the ASSGN statement used in these examples must not be used, and the corresponding operator command must be used in its place.

Figure 18. Cataloging Multiple Object Modules

CHAPTER 7. DATA SETS AND FILES

This chapter describes the nature and organization of data sets, the data management services provided by the Disk Operating System, and the ENVIRONMENT options used in file declarations to describe the data set to PL/I. It also explains how the compiled program produced by the compiler uses the data management services to create, modify, and access data sets. The term data set is synonymous with the term file as used extensively in the Disk Operating System for collections of data stored on an external storage medium; it is not synonymous with the PL/I term file, which is a PL/I data type given to identifiers that are associated with external data sets during program execution. Methods of creating and accessing data sets are given in Chapter 8 through Chapter 10.

Chapter 10 describes VSAM data sets. These differ significantly from other data set types; VSAM users will find that much of the information in Chapter 7 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 compiled program uses the data management routines of the operating system to create, update, and access data sets.

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. This name is specified in the DLBL job control statement. A data set on a magnetic tape device must have a name if the tape has IBM standard labels (see "Labels" on page 76). Data sets on punched cards, paper tape, unlabeled magnetic tape, or nonstandard labeled magnetic tape do not have names.

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" on page 73).

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 automatically by the data management routines.

Specify the record length 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 119.

Each character in the ASCII code is represented by a 7-bit pattern. 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

In a data set with fixed-length records, as shown in Figure 19 on page 74, all records have the same length. If the records are blocked, each block usually contains an equal number of fixed-length records (although the last block may be truncated). If the records are unblocked, each record constitutes a block.

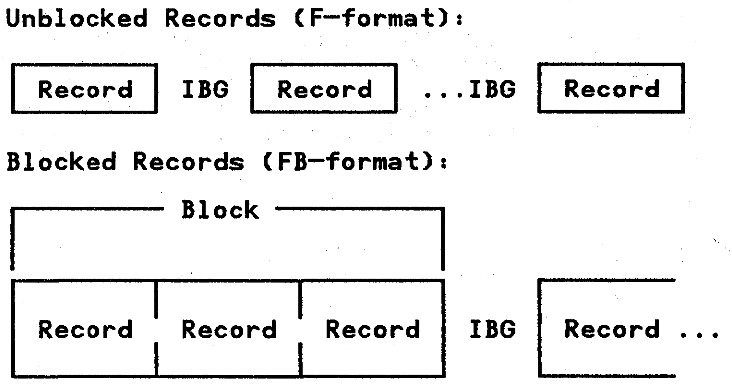


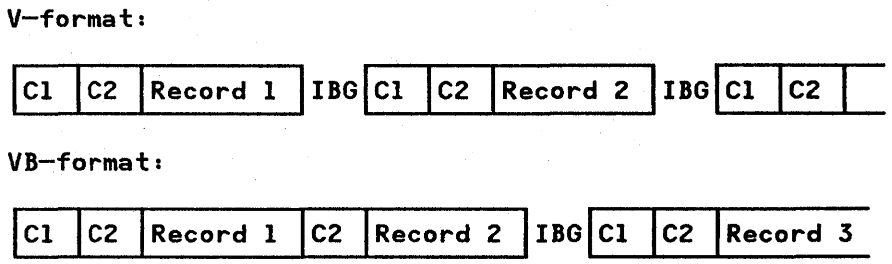
Figure 19. Fixed-Length Records

Variable-Length Records

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

V	Variable-length, unblocked
VB	Variable-length, blocked
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. Variable-length records are shown in Figure 20.



C1	Block control information
C2	Record or segment control information

Figure 20. 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.

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

- D-format records are similar to V-format records, except that the data they contain is recorded in ASCII.
- DB-format records are similar to VB-format records, 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 main types of non-VSAM data sets and the corresponding keywords describing their PL/I organization are:

Type of Data Set	**PL/I Organization**
Sequential	CONSECUTIVE
Indexed Sequential	INDEXED
Direct	REGIONAL
Virtual Storage	VSAM

Note: Do not confuse the operating system data set organizations "sequential" and "direct" with the PL/I file description attributes SEQUENTIAL and DIRECT, which describe how the file is to be processed by the PL/I program.

An option that describes the organization of the data set must be given in the ENVIRONMENT attribute, unless CONSECUTIVE organization is used. (CONSECUTIVE is assumed by default.)

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.

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, and addition 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 by a PL/I program in two ways: REGIONAL(1) and REGIONAL(3); in either case, the data set is divided into regions. Each region in a REGIONAL(1) data set contains one record only; a key specifying the region number identifies the record. Each region in a REGIONAL(3) data set occupies one track of a direct access device and contains one or more records; a key specifying both the region number and a key recorded with the record identifies a particular record within a particular region. Direct and sequential processing are both possible. Note that the DOS PL/I Optimizing Compiler does not support REGIONAL(2) data set organization.

VSAM data sets are described in Chapter 10.

LABELS

The operating system uses labels to identify magnetic tape and direct access volumes.

Magnetic tape volumes can have IBM standard or nonstandard labels, or they can be unlabeled. IBM standard labels have two parts: the initial volume label, and header and trailer labels. The initial volume label identifies a volume and its owner; the header and trailer labels precede and follow each data set on the volume. Header labels contain system information, device-dependent information (for example, recording technique), and data set characteristics. Trailer labels are almost identical with header labels, and are used when magnetic tape is read backward.

Direct access volumes have IBM standard labels. Each volume is identified by a volume label, which is stored on the volume. This label contains a volume serial number and the address of a volume table of contents (VTOC). The table of contents, in turn, contains a label, termed a data set label, for each data set stored on the volume.

DOS/VSE Tape Labels and DOS/VSE DASD Labels contain descriptions of the IBM standard magnetic tape and direct access data set labels, respectively.

DATA SETS AND FILES

A data set that is processed by a PL/I program must be represented within the program by a PL/I file. The declaration of a file in a PL/I program must include the ENVIRONMENT attribute, which describes the characteristics of the data set to be associated with the file; other file description attributes describe the type of processing to be performed by the file when the program is executed.

A PL/I file is not associated with a data set until the file is opened, and the association ceases when the file is closed. Consequently, through the TITLE option of the OPEN statement, the same file can be associated with different data sets during the execution of a single program, and the same data set can be accessed through different files.

JOB CONTROL STATEMENTS FOR DATA SETS

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

The DOS job control statements are:

ASSGN	identifies the device on which the data set will be mounted.
TLBL	labeled magnetic tape only; identifies the data set.
DLBL	direct access devices only; identifies the data set.
EXTENT	direct access devices only; defines the space to be occupied by the data set.
LBLTYP	allocates space for use by data management label-processing routines.

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

The ASSGN Statement

The Disk Operating System uses symbolic device names rather than actual device addresses to identify input/output devices within a program; accordingly, the MEDIUM option of the PL/I ENVIRONMENT attribute specifies only a symbolic device name and the type of device. It is the function of the ASSGN statement to relate a symbolic name to a device address; such a relationship is termed a device assignment.

Many device assignments are established permanently during initial program load. These are standard device assignments. If a program uses a standard device assignment for a data set, an ASSGN statement is not required for that data set. Therefore, you should always be aware of the standard assignments at your installation before running a program that processes a data set.

The TLBL Statement

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

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

The DLBL and EXTENT Statements

For each data set on a direct access device or diskette, there must be a DLBL statement and, usually, at least one EXTENT statement.

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

For a direct access device, the EXTENT statement defines the starting location and the size of the data set. The statement is not required for existing single-volume INPUT CONSECUTIVE data sets, since the information it contains can be retrieved from the data set label.

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

For a description of the DLBL and EXTENT statements when using the VSE/VSAM Space Management for SAM feature, see "VSE/VSAM Space Management for SAM Data Sets" on page 79.

The LBLTYP Statement

A LBLTYP statement should be included in any program that processes an INDEXED or REGIONAL data set or a data set on labeled magnetic tape. It requests the linkage editor to reserve storage for use by the data management label-processing routines.

The LBLTYP statement is not required when running on DOS/VSE with VSE/Advanced Functions.

DOS 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 data management, see DOS PL/I Optimizing Compiler: Execution Logic.)

The data management routines create and maintain data set labels and indexes. They transmit data between main and auxiliary storage, 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.

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 combination of data set organization and an access technique is termed an access method. The access methods used by the compiler and the subroutine libraries are:

SAM	Sequential Access Method
ISAM	Indexed Sequential Access Method
DAM	Direct Access Method
VSAM	Virtual Storage Access Method

The PL/I library subroutines use SAM for all stream-oriented transmission. They implement PL/I GET and PUT statements (other than GET or PUT STRING statements) by transferring the appropriate number of characters to or from the data management buffers, and use the data management routines to fill or empty the buffers.

Figure 21 on page 79 lists the access methods employed for record-oriented transmission.

Data Set Organization	File Attributes			Access Methods
CONSECUTIVE	SEQUENTIAL	INPUT OUTPUT UPDATE	BUFFERED or UNBUFFERED	SAM
INDEXED[1]	SEQUENTIAL	INPUT OUTPUT UPDATE	BUFFERED or UNBUFFERED	ISAM[2]
	DIRECT	INPUT UPDATE	—	ISAM[2]
REGIONAL	SEQUENTIAL	INPUT OUTPUT UPDATE	BUFFERED or UNBUFFERED	DAM[3]
	DIRECT	INPUT OUTPUT UPDATE	—	DAM[3]
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

[1]PL/I files declared with ENVIRONMENT(INDEXED) may be used to access VSAM key sequenced data sets (see Chapter 10).

[2]ISAM does not provide support for the 3350, 3330-11, 3375, 3380, or fixed block direct access storage devices.

[3]DAM does not provide support for the fixed block direct access storage devices.

Figure 21. Access Methods for Record-Oriented Data Transmission

VSE/VSAM SPACE MANAGEMENT FOR SAM DATA SETS

When using the VSE/VSAM Space Management for SAM feature, SAM data sets may be defined in VSAM space, either explicitly or implicitly. This applies to CONSECUTIVE RECORD files and STREAM files when a DASD device is specified in the MEDIUM option.

- For explicit defining, use Access Method Services to define a SAM ESDS with the required RECORDSIZE and RECORDFORMAT. Supply a DLBL statement for the file specifying VSAM. No EXTENT statement is needed.
- For implicit defining, supply a DLBL statement for the file specifying VSAM, and the RECORDS and RECSIZE parameters. The volume may be specified via an EXTENT statement or a default model for a SAM ESDS.

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

AUXILIARY STORAGE DEVICES

The following paragraphs state the record formats that are acceptable for various types of auxiliary storage devices, and summarize the salient operational features of these devices.

CARD READERS AND PUNCHES

The following card readers and punches can be used by PL/I programs:

IBM 1442 Card Read Punch
IBM 2501 Card Reader
IBM 2520 Card Read Punch
IBM 2540 Card Read Punch
IBM 2560 Card Read Punch
IBM 5425 Card Read Punch
IBM 3504 Card Reader
IBM 3505 Card Reader
IBM 3525 Card Punch

IBM 1442 Card Punch

The IBM 1442 Card Read Punch is functionally similar to the 2540 Card Read Punch, described below.

IBM 2501 Card Reader

The IBM 2501 Card Reader reads 80-column cards as 80-byte fixed-length EBCDIC records, and stacks the cards in a single stacker.

IBM 2520 Card Read Punch

The IBM 2520 Card Read Punch reads 80-column cards as 80-byte fixed-length EBCDIC records; it also punches fixed-length, variable-length, or unspecified-length EBCDIC records up to 80 bytes in length onto 80-column cards. The control bytes of variable-length records are not punched. Any attempt to block records is ignored. The 2520 has two stackers, one for cards that have been read, the other for cards that have been punched.

IBM 2540 Card Read Punch

The functions of the IBM 2540 Card Read Punch are similar to those of the 2520. In addition, the 2540 has five stackers. See Figure 22.

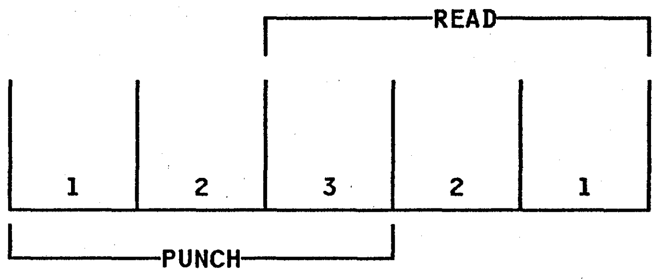


Figure 22. IBM 2540 Card Read Punch: Stacker Numbers

Two stackers are for cards that have been read, two are for cards that have been punched, and one is for cards that have been either read or punched. The stackers in each pair are numbered 1 and 2. The fifth is numbered 3. The stackers numbered 1 are normally used. For record-oriented files with the CTLASA or CTL360 ENVIRONMENT options, by inserting an ANS or machine code in the left byte of each record, stackers 2 and 3 can be used when punching cards.

IBM 2560 Card Read Punch

The following facilities of the IBM 2560 Card Read Punch are supported by PL/I:

CARD READING: The 2560 will read 80-column cards as fixed-length EBCDIC records up to 80 bytes in length, and stack the cards in a single stacker.

CARD PUNCHING: The 2560 will punch F-, V-, or U-format EBCDIC records up to 80 bytes in length onto 80-column cards. The control bytes of V-format records are not punched. Any attempt to block records is ignored. Stacker selection is available by means of CTLASA or CTL360 control characters. The CTLASA and CTL360 control characters are given in Chapter 9.

CARD PRINTING: The 2560 has up to six print heads, each of which prints a line of up to 64 characters on a card. F-, V-, and U-format records can be printed. The control bytes of V-format records are not printed.

A PL/I file that uses a 2560 must specify which function of the 2560 is being employed; that is, whether it is reading, punching, or printing. These functions are specified in the FUNCTION option of the ENVIRONMENT attribute. The FUNCTION option is also used to specify from which of the two input stackers cards are to be selected. The programmer can use the two input stackers independently to access two different data sets; there is no contention between cards from different stackers.

IBM 5425 Card Read Punch

The following facilities of the IBM 5425 Card Read Punch are supported by PL/I:

CARD READING: The 5425 will read 96-column cards as fixed-length EBCDIC records up to 96 bytes in length.

CARD PUNCHING: The 5425 will punch F-, V-, or U-format EBCDIC records up to 96 bytes in length onto 96-column cards. The control bytes of V-format records are not punched. Any attempt to block records is ignored. Stacker selection is available by means of CTLASA or CTL360 control characters. The CTLASA and CTL360 control characters are given in Chapter 9.

CARD PRINTING: The 5425 has up to 4 print heads, each of which prints a line of up to 32 characters on a card. F-, V-, and U-format records can be printed. The control bytes of V-format records are not printed.

A PL/I file that uses a 5425 must specify which of the functions of the 5425 is being used; that is, whether it is reading, punching, or printing. These functions are specified in the FUNCTION option of the ENVIRONMENT attribute. The FUNCTION option is also used to specify from which of the two input stackers cards are to be selected. The programmer can use the two input stackers independently to access two different data sets; there is no contention between cards from different stackers.

IBM 3504 and 3505 Card Readers

The IBM 3504 and 3505 Card Readers will read 80-column cards, and will provide, in addition to normal card reading, the following facilities:

- Optical Mark Read
- Read Column Eliminate
- Stacker Feature
- EBCDIC or Column Binary Modes

These additional features are described later in this chapter.

IBM 3525 Card Punch

The IBM 3525 Card Punch must be used in conjunction with an IBM 3505 Card Reader. However, these two devices are functionally separate and operate independently of each other.

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

These features are described in the following section. Note that a file that uses a 3525 must specify which function of the 3525 is to be employed, whether it is reading, punching, printing, or punch interpreting. These functions are specified in the FUNCTION option of the ENVIRONMENT attribute.

It is possible to use the 3525 to perform reading, punching, and printing operations on a single deck of cards. A separate PL/I file is used for each type of operation. Such files must be associated together as a group. This is achieved by the ASSOCIATE option of the ENVIRONMENT attribute. The use of associated files is discussed further in this chapter.

Features of the IBM 3504, 3505, and 3525

The following paragraphs describe the optional features of the IBM 3504, 3505, and 3525 devices.

OPTICAL MARK READING: The Optical Mark Read feature (OMR) is available only on the 3504 and 3505 Card Readers. This feature enables preprinted or pencil-written marks on a punched card to be read as data. The following rules apply:

- Data can be read in EBCDIC mode or in column binary mode.
- The associated PL/I file must have the attributes RECORD and INPUT and the OMR ENVIRONMENT option.
- The Read Column Eliminate (RCE) feature cannot be used in conjunction with the Optical Mark Read (OMR) feature.

- 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 a comma. 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 to 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.

- The symbolic device name SYSIPT must not be used for a file that uses the OMR feature.

- If columns 1 and 2 are used for optical character marks, a /* is not recognized as the file delimiter. Your PL/I program must then detect the last record on the file.

- When an OMR 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. Hence if the file is closed after the ENDFILE condition has been raised, that is, after a /* card has been read, a nondata card must follow the /*. If the file is closed before ENDFILE is raised, that is, without the /* being read, the /* card will suffice as a nondata card.

READ COLUMN ELIMINATE: The Read Column Eliminate (RCE) feature is optionally available on the 3504, 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:

- The symbolic device names SYSIPT and SYSRDR must not be used for a file that uses the RCE feature.
- 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 a comma. 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 RCE option must be declared in the ENVIRONMENT attribute for the file.
- The file can have either the STREAM or the RECORD attribute.
- The OMR feature cannot be used in conjunction with the Read column eliminate feature (RCE).
- If columns 1 and 2 are ignored, a /* is not recognized as the file delimiter. Your PL/I program must then detect the last record on the file.
- 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. Hence if the file is closed after the ENDFILE condition has been raised, that is, after a /* card has been read, a nondata card must follow the /*. If the file is closed before ENDFILE is raised, that is, without the /* being read, the /* card will suffice as a nondata card.

STACKER FEATURE: The stacker feature is optionally available on the 3504 and 3505, and is a standard feature on a 3525. For a 3504 and 3505, the stacker used for a particular file can be specified in the STACKER ENVIRONMENT option. For 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 STACKER ENVIRONMENT option.
- For record-oriented output files on 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.

The following rules apply to the use of the STACKER ENVIRONMENT option:

- The stacker feature cannot be used for a file that employs the card printing function of the 3525.
- If the value specified in the STACKER option is not 1 or 2, the UNDEFINEDFILE condition will be raised.
- The STACKER option is ignored if the file is device independent or if the use of stacker control characters is specified.
- If the STACKER option is used for an input file that uses OMR, any cards that contain unreadable mark data are stacked in the alternative stacker to that specified in the STACKER option.

EBCDIC OR COLUMN BINARY MODES: Cards processed by a 3504, 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 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 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 COLBIN ENVIRONMENT option must be specified.
- 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.
- The column binary feature cannot be used with a device independent file.

PRINTING ON CARDS: The card printing feature is available only on the IBM 3525 Card Punch. This feature is available in two forms:

- Two-line printing
- Multiline printing

Up to 64 characters can be printed on each line.

Stream- or record-oriented files can be used to print on cards. If, however, the file is associated with another file that uses the 3525 simultaneously, such as an INPUT or PUNCH file, the file must be record-oriented.

TWO-LINE PRINT FEATURE: If the 3525 has the two-line print feature and is used by a file with the PRINT attribute or a record-oriented file with the CTLASA or CTL360 ENVIRONMENT options, care should be taken to ensure that no attempt is made to print on any line other than lines 1 and 3 of the card. Such an attempt will cause the program to be terminated without raising a PL/I error condition. For a PRINT file, a maximum page size of 3 should be used to guard against this possibility. Files that do not have the PRINT attribute or the CTLASA or CTL360 ENVIRONMENT options will automatically print on lines 1 and 3. Note that SKIP(0) will cause the program to be terminated without raising a PL/I error condition. The MEDIUM ENVIRONMENT option must specify 3525T for a 3525 with the two-line print feature.

MULTIPLE PRINT FEATURE: If a 3525 with the multiline print feature is used, the file should have a maximum page size of 25, unless you intend that "pages" should span cards. 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. Note that SKIP(0) will cause the program to be terminated without raising a PL/I error condition. The MEDIUM ENVIRONMENT option must specify 3525 for a 3525 with the multiline print feature.

ASSOCIATED FILES ON THE 3525: If you use the ASSOCIATE option to associate two or more files for simultaneous use of a 3525, you should observe the following rules:

- The symbolic device names used for the associated files must not be any of these:

  ```
  SYSIPT
  SYSPCH
  SYSLST
  ```

- The files must all be record-oriented.

- Only a single buffer can be used on a read or punch file. Consequently, only BUFFERS(1) can be declared explicitly. One or two buffers can be declared for a print file.

- Files must be associated, and used, in the order:

  ```
  read -> punch -> print -> read ...
  ```

  This rule applies both to the specification of the ASSOCIATE option for each file and to the order of executing the input/output statements for the files. The print operation may be omitted or repeated; when it is employed, it must not cause any card feeding. The print operation, therefore, must not cause a new page to be started.

- The second operand of the FUNCTION option must specify all functions used by the associated files. Consequently, it must be identical in the declaration of each file in a group of associated files.

- If the system cannot open one of a group of associated files, it will be an error to attempt to use any of the other files in the group.

- If a file must be closed, all associated files must be closed as well, with no intervening I/O operation and no closed file in the group being reopened. If a file's last output statement was a LOCATE, an I/O operation takes place when the file is closed. This operation must be the next one in the normal I/O sequence, and must take place before any of the associated files is closed. The correct sequence of operations during the closure of a group of files can be ensured by using a multiple CLOSE statement with the files specified in the order read-punch-print.

- Closing a file may cause a card to be fed through the reader. Feeding takes place as follows.

Files in Group	**Files whose Closure causes Card Feed**
Read	Read (if RCE specified)
Punch	Punch
Print	Print
Read/Print	Print (if RCE specified on READ file)
Read/Punch/Print	Print
Read/Punch	Punch
Punch/Print	Print
PunchInterpret	Punch

No card feed operation is executed when a read or read print file for which RCE has not been specified is closed. To ensure that the last card in the file is fed through the reader, the file should not be closed until a read operation has raised the ENDFILE condition.

IBM 3800 Printing Subsystem

The IBM 3800 Printing Subsystem can be used in a manner compatible with IBM line printers; however, it can do more than line printers. For information on using its added capabilities, see DOS/VS IBM 3800 Printing Subsystem Programmer's Guide.

IBM 3881 Optical Mark Reader

The IBM 3881 Optical Mark Reader reads handwritten or machine-printed marks on paper documents. Documents are fed from a single hopper and are delivered to one of two stackers: the normal stacker (stacker 1) or the select stacker (stacker 2).

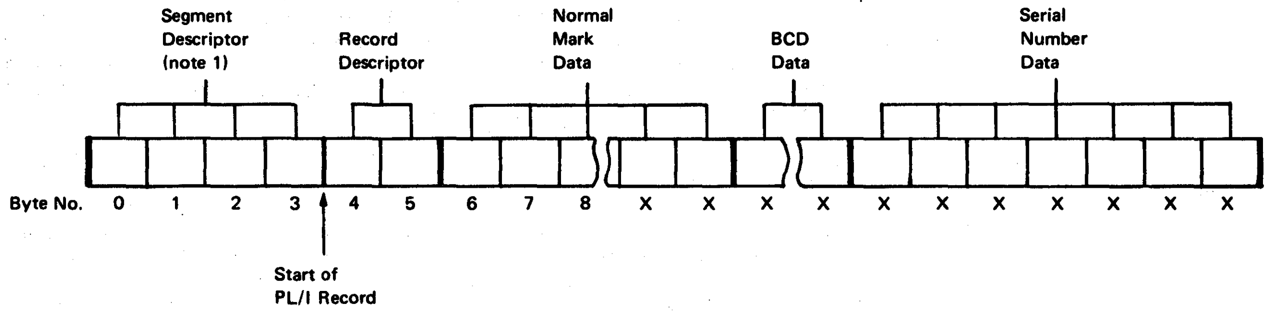
The contents of a document are transmitted to the PL/I program as a single record. The correspondence between marks on the document and the content of the output record is defined by means of a "format control sheet," which must be provided for each type of document that is to be read. Details of 3881 format control sheets are given in IBM 3881 Optical Mark Reader Models 1 and 2 Reference Manual and Operator's Guide.

The format control sheet defines sections of the document and specifies how the marks within these sections are to be interpreted. Each mark is translated into a numeric, alphabetic, or alphanumeric EBCDIC character. If the 3881 has the BCD feature, sections containing binary coded decimal data may also be defined. BCD data is translated to EBCDIC in the output record.

The 3881 can be equipped with a serial numbering feature that enables it to print a serial number on each document as it is read. If this feature is used, the serial number is given as part of the output record.

The format of a 3881 output record is shown in Figure 23 on page 88, together with the meaning of the record descriptor bytes. Note that the first four bytes of the record are not transmitted to the PL/I program.

The output record of the 3881 is effectively V-format, and may be read into a varying length string. The PL/I file that is used to access the data set should have a block size or record length specified that is at least 4 bytes larger than the record actually read. This is to allow for the segment descriptor word. Any record format may be specified, but the file will be treated by the compiler as variable-length unblocked. If neither block size nor record length is specified, a default of block size equals 900 is taken. A block size of 900 is also the largest block size that may be specified.



Byte 4		Conditions			
Hex	EBCDIC	Serial Numbering Used	Dcmt sent to Select Stacker	Reject Characters on Dcmt (note 2)	BCD Errors on Dcmt (note 2)
C8	H	No	No	No	No
C1	A	No	No	No	Yes
C2	B	No	No	Yes	No
C3	C	No	No	Yes	Yes
C4	D	No	Yes	No	No
C5	E	No	Yes	No	Yes
C6	F	No	Yes	Yes	No
C7	G	No	Yes	Yes	Yes
F0	0	Yes	No	No	No
F1	1	Yes	No	No	Yes
F2	2	Yes	No	Yes	No
F3	3	Yes	No	Yes	Yes
F4	4	Yes	Yes	No	No
F5	5	Yes	Yes	No	Yes
F6	6	Yes	Yes	Yes	No
F7	7	Yes	Yes	Yes	Yes

Byte 5		Condition (note 3)
Hex	EBCDIC	
F0	0	Basic format
F1	1	Format 1
F2	2	Format 2
F3	3	Format 3
F4	4	Format 4
F5	5	Format 5

Notes:
1. The segment Descriptor is not passed to the PL/I program.
2. Invalid combinations of marks cause a special code to be placed in the corresponding position in the output record. This code is either hex '3F' or optionally hex '7C' which is a printable character.
3. The format type is as specified on the last format control sheet read by the 3881.

Figure 23. Format of IBM 3881 Output Records

LINE PRINTERS

The printer accepts F-, V-, 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.

When using a record-oriented file, you can control the printer line spacing dynamically by inserting an ANS or machine code in the first byte of each record; indicate which code you are using in the ENVIRONMENT attribute (CTLASA or CTL360 option). The control character is not printed. If you do not specify the line spacing, single spacing (no blanks between lines) is assumed.

MAGNETIC TAPE

Magnetic tape devices accept ASCII, fixed-length, variable-length, and undefined-length records.

Magnetic tape can be used in a number of different ways, according to the features available or that are required for a particular program. You should find out what the standard features for the magnetic tape devices in your installation are, and use the job control ASSGN statement to specify any of the features that are nonstandard. The features that are available are described below.

Track Width

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.

Translation Feature

The translation feature changes character data from EBCDIC (the 8-bit code used in IBM operating systems) to BCD (the 6-bit code used on 7-track tape) or vice versa.

Conversion Feature

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.

You can specify fixed-, variable-, or undefined-length records for 9-track magnetic tape, but fixed-length only for 7-track magnetic tape, 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.)

Recording Density

The maximum recording density available depends on the model number of the tape drive that you use.

For 9-track tape, track density is controlled by the ASSGN statement. Refer to DOS/VSE System Control Statements.

For 7-track tape, the standard recording density for both types of drive unit is 200 bytes per inch; you can use the ASSGN statement to select alternatives of 556 or 800 bytes per inch.

Note: When a read or write hardware error occurs on a magnetic tape device with short records (12 bytes on a read and 18 bytes on a write), these records will be ignored.

Magnetic Tape Volumes with Multiple Data Sets

A magnetic tape volume can contain more than one data set. Consequently, it may be necessary to wind (or rewind) the volume to a particular position before processing can take place. The format of both labeled and unlabeled data sets on magnetic tape volumes is shown in Figure 24 on page 90. Note that a tapemark is used to separate a data set label from the preceding and following data sets, and to separate unlabeled data sets.

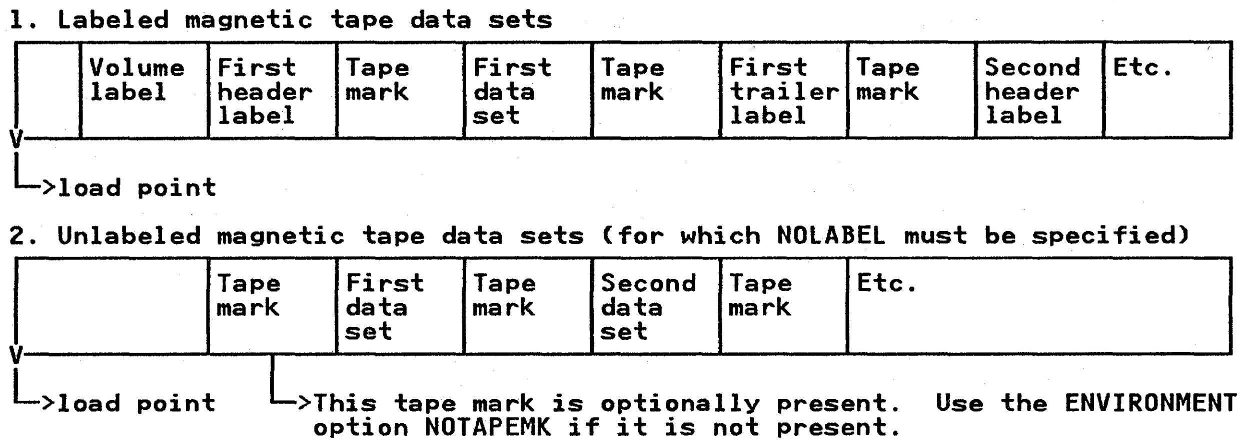

1. Labeled magnetic tape data sets
Volume label
First header label
Tape mark
First data set
Tape mark
First trailer label
Tape mark
Second header label
Etc.
>load point
2. Unlabeled magnetic tape data sets (for which NOLABEL must be specified)
Tape mark
First data set
Tape mark
Second data set
Tape mark
Etc.
>load point
>This tape mark is optionally present. Use the ENVIRONMENT option NOTAPEMK if it is not present.

Figure 24. Format of Magnetic Tape Volumes

If a program is to access or create a data set that is not the first data set on the volume, the MTC job control statement causes the volume to be positioned at the required data set (or its header or trailer label) before the program is invoked. After invocation of the program, the program itself must control the positioning of the volume. This is especially important if the program will reopen, for reading backward, a file that it has just closed, or if it will open a file associated with a data set to follow the data set it has just processed. For this purpose, the LEAVE option of the ENVIRONMENT attribute and of the CLOSE statement is available. On closing the file, the effect of the LEAVE option is to suppress the automatic rewinding of the volume.

Magnetic Tape Labels

Data sets on magnetic tape can have IBM standard labels or nonstandard labels, or they can be unlabeled.

IBM standard labels are processed by the operating system, provided that a program is link-edited with a LBLTYP job control statement to reserve space for label processing, and that a TLBL statement is included for each data set. The LBLTYP statement may not be required, dependent upon the level of your operating system.

You can use a PL/I program to process IBM standard or nonstandard labels by treating them as separate data sets. The ENVIRONMENT attribute for the associated files must include the option NOLABEL. To skip a label, use the MTC job control statement. (Refer to DOS/VSE System Control Statements.) Multivolume data sets cannot be unlabeled if they are to be created or accessed automatically.

An unlabeled data set requires no special treatment.

Backward Processing of Magnetic Tape Data Sets

If a PL/I program opens an input file with the BACKWARDS attribute, the volume containing the associated data set must be positioned at the trailer label of this data set. If the data set was created or retrieved in a forward direction immediately before it is to be retrieved in reverse order, and if the LEAVE option was specified when the output file was closed, the volume will be positioned correctly for the input file to be opened.

Otherwise, the MTC job control statement must be used to position the volume correctly.

Use of the ENDFILE Condition

A data set in a magnetic tape volume with multiple data sets can be accessed, and then either the same data set can be accessed in reverse order or the next data set in the volume can be accessed. However, the LEAVE option must be specified for the first file, and the ENDFILE condition must be raised before this file is closed. This ensures that the magnetic tape volume is positioned correctly, after the tapemark, when the second file is opened.

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, a 3375 direct access storage device has 12 recording surfaces, each of which has 946 concentric tracks; thus, it contains 946 cylinders, each of which includes 12 tracks.

When you create a data set on a direct access device, you must always indicate to the operating system, in one or more EXTENT statements, how much auxiliary storage the data set requires. You must specify the space requirement in terms of tracks or blocks. 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.

For detailed information that will enable you to determine the amount of space you will require, see the publications for the direct access device you will be using.

IBM 3540 Diskette Input/Output Unit

Diskettes of the IBM 3540 Input/Output Unit accept fixed-length, unblocked records.

The storage space on a diskette consists of 77 tracks, track 0 through track 76. Of these tracks, 73 are available to you; the remaining tracks are used by the system (track 0 is used to hold up to 19 file labels, each of which describes an individual file; three tracks are reserved).

Tracks are divided into predefined sections called sectors. There are 26 sectors, numbered 01 through 26, on each track. Each sector holds one record. A record is a collection of related items of data; standard record length for the 3540 is 128 characters. One diskette holds 1898 records.

ASSOCIATING DATA SETS WITH FILES

A file in a PL/I program is generally associated with a physical data set by means of options of the ENVIRONMENT attribute. The MEDIUM option must be specified in every file declaration. The MEDIUM option must specify a 'symbolic device name' and, optionally, a 'device type' for the data set. The 'symbolic device name' is used by the operating system to identify a particular input/output device. It takes the form 'SYSxxx' where xxx is the range 000 to 255, or, for IBM standard system devices, is SYSRDR, SYSIPT, SYSIN, SYSOUT, SYSLST, SYSPCH, SYSLNK, and SYSLOG. For a particular system, the symbolic device names are assigned to particular physical devices. However, these assignments can be changed temporarily by the job control ASSGN statement. Except for the special circumstances given later in this chapter under the "MEDIUM Option," the device type must be specified in the source program so that the compiler can cause the appropriate data management routines to be link-edited.

The device type of the MEDIUM option must correspond to the physical input/output unit type assigned to the file either automatically or by using the ASSGN statement. For example, if the device type indicates magnetic tape, the file must be assigned to a magnetic tape unit.

Consider the following example:

```
  DECLARE MASTER FILE RECORD INPUT
          SEQUENTIAL ENVIRONMENT
  (...MEDIUM(SYS006,2400)...  );
```

In this declaration, the file MASTER is assigned the symbolic device name SYS006 and the device type for MASTER is declared to be an IBM 2400 Magnetic Tape Unit.

If the operating system in which the above file declaration is used automatically associates the name SYS006 with a suitable magnetic tape unit, no further assignment is necessary. Otherwise, the ASSGN statement must be used to assign a system file to a magnetic tape unit. The job control program used with the execution module must contain the statement partially shown below:

```
  // ASSGN SYS006,...
```

This statement associates the logical device name SYS006 with a physical input/output unit which must, of course, be a magnetic tape. The specific tape unit to be used follows the SYS006 on the ASSGN statement.

The MEDIUM option only associates a file with a device or volume; the first seven characters of the file name or the identifier specified in a TITLE option associates a file with the job control statements DLBL or TLBL.

If the TITLE option of the OPEN statement is specified, the filename is converted to a character string, when necessary. The first seven characters of the filename identify the data set (via the filename of a DLBL or TLBL job control statement). If the option is not specified, the first seven characters of the filename (padded or truncated) are taken to be the DLBL or TLBL filename. This filename is used when a file is passed as a parameter, rather than the external filename, which is truncated in a different manner.

If the file expression used in an OPEN statement is a file constant, then a TLBL or DLBL statement must name the file constant. If the OPEN statement does not specify a file constant, then a TLBL or DLBL statement must name the value of the file expression. For example:

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

A TLBL statement associates the data set STOCK with the file constant RPRICE, thus:

```
  // TLBL RPRICE,'STOCK'
```

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.....,
          .
          .
        F=A;
          .
          .
LAB:    READ FILE (F) .....;
          .
          .
          .
        F=B;
        GO TO LAB;
```

The READ statement is used to read files A and B. Note that files A and B remain open after the READ statement has been executed in each instance.

The following OPEN statement illustrates the TITLE option:

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

If this statement is to be executed, there must be a TLBL or DLBL job control statement in the current job step with DETAIL1 as its file name. It might appear as follows:

```
// TLBL DETAIL1,'DETAILA'
```

Thus, the data set DETAILA is associated with the file DETAIL through the TLBL file name, DETAIL1.

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 data set is taken to be MASTER1A. After processing, the file is closed, dissociating the file name and the data set. During the second iteration of the do group, MASTER is opened again. This time, MASTER is associated with the data set MASTER1B. Similarly, during the final iteration of the do group, MASTER is associated with the data set 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 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.

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 within parentheses in an option list; the syntax is:

Syntax

```
ENVIRONMENT(option-list)
```

Abbreviation: ENV

A constant or variable can be used with most of those ENVIRONMENT options that require decimal integer arguments, such as block sizes and record lengths. The options EXTENTNUMBER and BUFFERS require constants. The variable must be unsubscripted and unqualified with the attributes, FIXED BINARY(31,0) and STATIC.

The options may appear in any order, and are separated by blanks. The options themselves cannot contain blanks.

The following example illustrates the syntax of the ENVIRONMENT attribute in the context of a complete file declaration. (The options specified are for VSAM and are discussed in Chapter 10.)

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

Figure 25 on page 95 and Figure 26 on page 96 summarize the PL/I file attributes, the ENVIRONMENT options, and certain qualifications on the use of both.

DATA SET ORGANIZATION OPTIONS

The options that specify data set organization are:

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

Each option is described in the chapter applicable to its data set organization. If the data set organization option is not specified in the ENVIRONMENT attribute, CONSECUTIVE is assumed by default.

ENVIRONMENT OPTIONS

This chapter describes the options that apply to two or more data set organizations:

```
F|FB|V|VB|D|DB|U    VERIFY
GENKEY              EXTENTNUMBER
MEDIUM              COBOL
RECSIZE             SCALARVARYING
BLKSIZE             KEYLENGTH
BUFFERS
```

Each remaining ENVIRONMENT option is described in the chapter pertaining to its data set organization.

Types of File		Record									
		Sequential						Direct			
		CONSE-CUTIVE			REGIONAL						
PL/I File Attributes	Stream	Buffered	Unbuffered	INDEXED	Buffered	Unbuffered	VSAM	VSAM	INDEXED	REGIONAL	Attributes Implied
BACKWARDS[3]	—	O	O	—	—	—	—	—	—	—	FILE RECORD SEQUENTIAL INPUT
BUFFERED	—	D	—	D	D	—	D	S	—	—	FILE RECORD SEQUENTIAL
DIRECT	—	—	—	—	—	—	S	S	S	S	FILE RECORD KEYED
ENVIRONMENT	I	S	S	S	S	S	S	S	S	S	FILE
FILE	I	I	I	I	I	I	I	I	I	I	
INPUT[1]	D	D	D	D	D	D	D	D	D	D	FILE
KEYED[4]	—	—	—	O	O	O	O	O	I	I	FILE RECORD
OUTPUT	O	O	O	O	O	O	O	O	—	O	FILE
PRINT[1]	O	—	—	—	—	—	—	—	—	—	FILE STREAM OUTPUT
RECORD	—	I	I	I	I	I	I	I	I	I	FILE
SEQUENTIAL	—	D	D	D	D	D	D	D	—	—	FILE RECORD
STREAM	D	—	—	—	—	—	—	—	—	—	FILE
UNBUFFERED	—	—	S	—	—	S	S	D	—	—	FILE RECORD SEQUENTIAL
UPDATE[2]	—	O	O	O	O	O	O	O	O	O	FILE RECORD

[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 tape files.
[4] KEYED is required for INDEXED and REGIONAL output.

Key:

I	Must be specified or implied
D	Default
O	Optional
S	Must be specified
—	Invalid

Figure 25. Attributes of PL/I File Declarations

Record Format Options for Record-Oriented Data Transmission

Record formats supported depend on the data set organization.

Syntax

F|FB|V|VB|D|DB|U

Records can have one of the following formats:

Fixed-length	F	unblocked
	FB	blocked
Variable-length	V	unblocked
	VB	blocked
	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.

Types of File / ENVIRONMENT Options	Stream	Record									
		Sequential							Direct		
		CONSE-CUTIVE			REGIONAL						
		Buffered	Unbuffered	INDEXED	Buffered	Unbuffered	VSAM		VSAM	INDEXED	REGIONAL
ADDBUFF	–	–	–	–	–	–	–		–	O	–
ASCII	O	O	–	–	–	–	–		–	–	–
ASSOCIATE	–	S	S	–	–	–	–		–	–	–
BKWD	–	–	–	–	–	–	O		–	–	–
BLKSIZE	I	I	I	I	I	I	N		N	I	I
BUFFERS	I	I	–	I	I	–	N		N	–	–
BUFND	–	–	–	–	–	–	O		O	–	–
BUFNI	–	–	–	–	–	–	O		O	–	–
BUFOFF	O	O	–	–	–	–	–		–	–	–
BUFSP	–	–	–	–	–	–	O		–	–	–
CMDCHN	I	I	I	–	–	–	–		–	–	–
COBOL	–	O	O	O	O	O	O		O	O	O
COLBIN	–	S	S	–	–	–	–		–	–	–
CONSECUTIVE	D	D	–	–	–	–	O		O	–	–
CTLASA|CTL360	–	O	O	–	–	–	–		–	–	–
EXTENTNUMBER	–	–	–	I	I	I	–		–	I	I
F|FB	–	–	–	S	–	–	N		N	S	–
F|FB|D|DB|U	S	S	–	–	–	–	N		N	–	–
F|FB|V|VB|U	I	S	S	–	–	–	N		N	–	–
F|U	–	–	–	–	S	S	N		N	–	S
FILESEC	I	I	I	–	–	–	–		–	–	–
FUNCTION	O	O	–	–	–	–	–		–	–	–
GENKEY	–	–	–	O	–	–	O		O	–	–
GRAPHIC	O	–	–	–	–	–	–		–	–	–
HIGHINDEX	–	–	–	O	–	–	–		–	O	–
INDEXAREA	–	–	–	–	–	–	–		–	O	–
INDEXED	–	–	–	S	–	–	O		O	S	–
INDEXMULTIPLE	–	–	–	O	–	–	–		–	O	–
KEYLENGTH	–	–	–	S	S	S	C		C	S	S
KEYLOC	–	–	–	O	–	–	–		–	O	–
LEAVE	O	O	O	–	–	–	–		–	–	–
MEDIUM	I	S	S	S	S	S	N		N	S	S
NOFEED	I	I	I	–	–	–	–		–	–	–
NOLABEL	O	O	O	–	–	–	–		–	–	–
NOTAPEMK	O	O	–	–	–	–	–		–	–	–
NOWRITE	–	–	–	–	–	–	–		–	O	–
OFLTRACKS	–	–	–	O	–	–	–		–	–	–
OMR	–	S	S	–	–	–	–		–	–	–
PASSWORD	–	–	–	–	–	–	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

<u>Comments</u>

Either BLKSIZE or RECSIZE or both must be specified for CONSECUTIVE, INDEXED, and REGIONAL files.

For ASCII data sets, only F,FB,D, DB, and U are valid.

CTLASA/CTL360 not valid for ASCII data sets.

KEYLENGTH not valid for REGIONAL(1) files.

Only F valid for REGIONAL(1) files.

NOWRITE is valid only for UPDATE files.

GENKEY is valid only for INPUT or UPDATE files; KEYED is required.

Figure 26 (Part 1 of 2). Options of PL/I File Declarations

Types of File		Record									
		Sequential						Direct			
		CONSE-CUTIVE			REGIONAL						
ENVIRONMENT Options	Stream	Buffered	Unbuffered	INDEXED	Buffered	Unbuffered	VSAM	VSAM	INDEXED	REGIONAL
RCE	I	S	S	—	—	—	—	—	—	—
RECSIZE	I	I	I	I	I	I	C	C	I	I
REGIONAL	—	—	—	—	S	S	—	—	—	S
REUSE	—	—	—	—	—	—	O	O	—	—
SCALARVARYING	—	O	O	O	O	O	O	O	O	O
SKIP	—	—	—	—	—	—	O	—	—	—
STACKER	I	S	S	—	—	—	—	—	—	—
UNLOAD	O	O	O	—	—	—	—	—	—	—
VERIFY	O	O	O	O	O	O	—	—	O	O
VOLSEQ	I	I	I	—	—	—	—	—	—	—
VSAM	—	—	—	—	—	—	S	S	—	—
WRTPROT	I	I	I	—	—	—	—	—	—	—

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 26 (Part 2 of 2). Options of PL/I File Declarations

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.

Record Format Options for Stream-Oriented Data Transmission

The record format options for stream-oriented data transmission are discussed in Chapter 8.

GENKEY Option

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 READ KEY, 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 because 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 want 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 KEYLENGTH option 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).

MEDIUM Option

The MEDIUM (device type) option of the ENVIRONMENT attribute must be specified for each file declaration except for VSAM files, in which case it is ignored if specified.

Syntax

```
MEDIUM(symbolic-device-name
      [,device-type])
```

The symbolic device name specification has the form SYSxxx, where xxx may be:

IPT	system input device
LST	system output device used for listing
PCH	system output device (card punch)
000-255	symbolic device names SYS000 through SYS255

The device type specification contains the number of the device to be used. For instance, if the IBM 1442N1 Card Read/Punch is to be used, the option would be written as 1442.

Device Type	Number	MEDIUM Specification
Card Readers and Punches	IBM 2540	2540
	IBM 2560	2560
	IBM 1442N1	1442
	IBM 1442N2	1442
	IBM 2520B1	2520
	IBM 2520B2	2520
	IBM 2520B3	2520
	IBM 2501	2501
	IBM 3504	3504
	IBM 3505	3505
	IBM 3525	
	(multi-line print)	3525
	(2-line print)	3525T
	IBM 3881	3881
	IBM 5425	5425
Printers	IBM 1403	1403
	IBM 1404	1404
	IBM 1443	1443
	IBM 1445	1445
	IBM 3211	3211
	IBM 5203	5203
	IBM 3203	3203
Magnetic Tape Drives	IBM 2400 (9-track)	2400
	IBM 2400 (7-track)	2400
	IBM 3410/3411	3410
	IBM 3420	3420
	IBM 8809	2400
DASD	IBM 2311	2311
	IBM 2314	2314
	IBM 2321	2321
	IBM 3330	3330
	IBM 3340	3340
	IBM 3350	3350
	IBM 3375	2311
	IBM 3380	2311
	IBM fixed block devices	FBA
Diskette Unit	IBM 3540	3540

Figure 27. Device Types and Corresponding Specifications

Figure 27 shows how the individual device types are specified. The device type specification is optional in certain circumstances, but if it is specified in the MEDIUM option, it can subsequently be overwritten only under the conditions described below.

- The 3330-11, 3350, 3375, and 3380 devices are not supported for ISAM files with INDEXED organization. When 3330 is specified, the compiler assumes the 3330 model 1 device is meant.
- The fixed block devices are not supported for files with INDEXED or REGIONAL organization; therefore, for these types of files, device type in the MEDIUM option must not specify FBA.

The device types listed in Figure 27 may be assigned to the symbolic device names SYSIPT, SYSLST, and SYSPCH as shown in Figure 28.

Symbolic Device Name	**Device Type**	
SYSIPT	IBM 2540 (reader) IBM 2560 (reader) IBM 1442N1 IBM 2501 IBM 2520B1 IBM 2400 (7- or 9-track) IBM 2311 IBM 2314 IBM 3504 IBM 3505 IBM 3525 IBM 3540	IBM 3420 IBM 3410/3411 IBM 3330 IBM 3340 IBM 3350 IBM 3375 IBM 3380 IBM fixed block devices IBM 3881 IBM 5425 IBM 8809
SYSLST	IBM 1403 IBM 1404 IBM 1443 IBM 2400 (7- or 9-track) IBM 2311 IBM 2314 IBM 3211 IBM 3420 IBM 3410/3411 IBM 3540	IBM 3330 IBM 3340 IBM 3350 IBM 3375 IBM 3380 IBM fixed block devices IBM 3881 IBM 5203/3203 IBM 8809
SYSPCH	IBM 2540 (punch) IBM 2560 (punch) IBM 1442N1 IBM 1442N2 IBM 2520B1 IBM 2520B2 IBM 2520B3 IBM 2400 (7- or 9-track) IBM 2311 IBM 2314 IBM 3525 IBM 3540	IBM 3420 IBM 3410/3411 IBM 3330 IBM 3340 IBM 3350 IBM 3375 IBM 3380 IBM fixed block devices IBM 3881 IBM 5425 IBM 8809

Figure 28. Device Types Associated with SYSIPT, SYSLST, and SYSPCH

The operating system input/output facilities use the symbolic device name to associate the PL/I file with a symbolic device name used by the operating system. Within the system, the symbolic device name is assigned, either permanently or temporarily, to a device. However, the compiler requires to know the device type used for a data set associated with a PL/I file, so that the correct data management input/output routines can be incorporated into the object program by the linkage editor. (For more information, see "Associating Data Sets with

Files" on page 91.) Note that the choice of device type should normally be made prior to compilation, and that alteration will usually require recompilation with a modified MEDIUM option. However, a limited degree of device independence is possible.

A device type need not be specified in the MEDIUM option for the symbolic device names SYSIPT, SYSLST, and SYSPCH. By means of the job control ASSGN statement, the user can specify the actual device type to be used during execution of the program. If you do not know which direct access device will be used at execution time, you should code 2311. If you will be using tape but do not know which tape device will be used at execution time, you should code 2400. After release 3 of DOS VSE/Advanced Functions, some device substitution for non-ISAM files is allowed.

For a device such as the IBM 3375, the MEDIUM option could specify a 2311. When the program executes with a SIZE parameter on the EXEC statement, the system will use the device that is assigned to the logical unit.

When device substitution is used, it must be a device of the same class. If 2311 is specified in the MEDIUM option, it must be assigned to a disk device. Likewise, 2400 implies a magnetic tape drive. Earlier DOS systems only supported device independence for the 3330-11, 3350, or FBA devices.

The tables and routines necessary for device independence are generated by the compiler if the following requirements are met:

- The output file SYSLST must either have the STREAM and the PRINT attribute or the RECORD attribute with the CTLASA option. The output file SYSPCH must have the RECORD attribute with the CTLASA option. The first character of each record is interpreted as a control character in these cases. Input files (SYSIPT) may be either STREAM or RECORD. In all cases, RECORD files must have the BUFFERED attribute.

  Note: For the output file SYSPCH, if the device specified in the ASSGN statement is an IBM 2560, the FUNCTION option (described under "CONSECUTIVE Data Sets" on page 119) cannot be specified on the device independent file declaration; stacker selection is made through the ASSGN statement, and read and punch operations are performed for input and output files respectively. Only the "V" and "W" ANS control characters can be used with device independent files for output through the IBM 2560.

- The IBM 5425 multifunction card unit uses 96-column cards; when assigned to SYSIPT or SYSPCH, only 80 columns of data may be read or punched. In order to read or punch the full 96 columns, the MEDIUM option must specify 5425.

- Records must be unblocked and of fixed length. The maximum record length is as follows:

  80 for SYSIPT
  121 for SYSLST
  81 for SYSPCH

Note: If the data set is to reside on a direct access device, the record sizes 80, 121, and 81 must be the values for SYSIPT, SYSLST, and SYSPCH respectively.

If the above requirements are met, any device type supported for the respective symbolic-device-name (see Figure 28 on page 100) may be assigned without the necessity of recompilation.

RECSIZE Option

The RECSIZE option specifies the record length.

```
── Syntax ──────────────────────────────
RECSIZE(record-length)
```

For files other than those files associated with VSAM data sets, the 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 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 length should be specified.

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-, and undefined-length (except ASCII data sets): 32,760 bytes

Variable-length: 32,756 bytes

ASCII data sets: 9999

VSAM data sets: 32,761 bytes

Zero value:
Default action is taken (see "Record Format, BLKSIZE, and RECSIZE Defaults" on page 103).

Negative Value:
The UNDEFINEDFILE condition is raised.

BLKSIZE Option

The BLKSIZE option specifies the maximum block size on the data set. It does not apply to VSAM data sets, and is ignored if specified.

```
── Syntax ──────────────────────────────
BLKSIZE(block-size)
```

The block size is the sum of:

1. The total length(s) of one of the following:

   A single record
   Several records

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

2. Any further control bytes required. Variable-length blocked records require 4 for the block size; fixed- and undefined-length records do not require any.

   or

   Any block prefix bytes required (ASCII data sets only).

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

The block size value 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)

Zero value:
Default action is taken (see "Record Format, BLKSIZE, and RECSIZE Defaults")

Negative value:
The UNDEFINEDFILE condition is raised

The relationship of the block size to the record length depends on the record format:

FB-format:
The block size must be a multiple of the record length.

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

1. The maximum length of any record
2. Four control bytes

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)

Record Format, BLKSIZE, and RECSIZE Defaults

If, for a non-VSAM data set, any of the record format options is not specified, the following action is taken:

Record Format:	An error message is produced.
Block size or record length:	If one of these is specified, a value for the other 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.

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.

Syntax

```
BUFFERS({1|2})
```

The BUFFERS option specifies the number of buffers to be allocated for a data set; this number can only be one or two. For CONSECUTIVE or INDEXED RECORD files with sequential access, two buffers are allocated by default; the user can change this to one. For all other record files, a default of one, which cannot be changed by the user, is allocated.

VERIFY Option

The VERIFY option is used to specify that a read-check is to be performed after every write operation. This option is permitted only with direct access devices. It is assumed for a 2321.

Syntax

```
VERIFY
```

EXTENTNUMBER Option

The EXTENTNUMBER option is used to specify the number of extents used for REGIONAL or INDEXED data sets.

Syntax

```
EXTENTNUMBER(n)
```

For REGIONAL data sets, EXTENTNUMBER(n) is optional. If specified, n must be greater than 0 and less than 256. The default value of n is 1.

For INDEXED data sets, EXTENTNUMBER(n) is optional. If specified, the value for n must include all data area extents, the master index and cylinder index extents (which must be adjacent to one another), and all independent overflow extents. Master and cylinder index extents count as one extent, although each index requires a separate EXTENT job control statement. Thus, the minimum number that can be specified is two: one extent for one prime data area and one for the cylinder index. The maximum value for n is 255, and the default value is two.

COBOL Option

The COBOL (data interchange) option specifies that structures in the data set associated with the file will be mapped as they would be in a COBOL compiler.

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 filename 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, and 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.

See Chapter 15, "Communication with COBOL, FORTRAN, and RPG" on page 247, for supported COBOL compilers and for PL/I equivalents of COBOL data types.

SCALARVARYING Option

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 8. 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.

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. Because stream-oriented transmission always treats the data in a data set as a continuous stream, it can be used only to process data sets with CONSECUTIVE organization.

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 the data items from the program variables into character or graphic 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.

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.[1] For example, the Kanji print utility does not allow graphic strings to be continued onto another line.

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(option-list);
```

Default file attributes are shown in Figure 25 on page 95; these attributes are described in the OS and DOS PL/I Language Reference Manual. The PRINT attribute is described further in this chapter.

ENVIRONMENT OPTIONS

The following options are applicable to stream-oriented data transmission and are described in this chapter:

```
CONSECUTIVE
F|FB|V|VB|D|DB|U
RECSIZE
GRAPHIC
```

[1] Details on processing Japanese or Chinese graphics are available through the IBM World Trade Americas/Far East Corporation.

The following options also apply to stream-oriented data transmission:

ASCII	NOFEED
BLKSIZE	NOLABEL
BUFFERS	NOTAPEMK
BUFOFF	RCE
CMDCHN	STACKER
FILESEC	UNLOAD
FUNCTION	VERIFY
LEAVE	VOLSEQ
	WRTPROT

Options that apply to two or more data set organizations are described in Chapter 7. Each remaining option is described in the chapter pertaining to its data set organization. Figure 26 on page 96 summarizes the ENVIRONMENT options.

CONSECUTIVE Option

STREAM files must have CONSECUTIVE data set organization. The CONSECUTIVE option for STREAM files is the same as that described under "CONSECUTIVE Data Sets" on page 119 in Chapter 9.

Syntax

CONSECUTIVE

CONSECUTIVE is the default data set organization.

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|V|VB|D|DB|U

Records can have one of the following formats, as described in Chapter 7.

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

Blocking and deblocking of records is performed automatically.

RECSIZE Option

RECSIZE for stream-oriented data transmission is the same as that described in Chapter 7. 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 options are not specified in the ENVIRONMENT attribute, the following action is taken:

INPUT files:

If the symbolic device name SYSIPT or a unit record device is specified in the MEDIUM option, the defaults are F for record format and 80 for record length, block size, and LINESIZE. For all other INPUT files, defaults are applied as for record-oriented data transmission, described under "Record Format, BLKSIZE, and RECSIZE Defaults" in Chapter 7.

OUTPUT files:

If the symbolic device name SYSPCH or a card punch is specified in the MEDIUM option, the defaults are F for record format, 80 for LINESIZE, and 81 for both record length and block size. If the symbolic device name SYSLST or a line printer is specified, the defaults are F for record format, 120 for LINESIZE, and 121 for both record length and block size. For all other 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, or U:	line size + 1
V, VB, D, or DB:	line size + 5

Non-PRINT files:

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

Block size:

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

Buffer offset:

F, FB, or U:	0
D or DB:	4

Notes:

1. For PRINT files, the default LINESIZE value is 120. There is no default LINESIZE for non-PRINT files.

2. If the block size as calculated above is greater than 32,760, the block size is set to the record length + 4, and the record format is set to V. For an ASCII data set, if the default block size is greater than 32,700 (or 9999 if BUFOFF is specified without a prefix length value), the

block size is set to record length + block prefix and the record format is set to D.

3. 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.

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.

CREATING A DATA SET FOR STREAM-ORIENTED DATA TRANSMISSION

To create a data set, you must give the operating system certain information in your PL/I program.

ESSENTIAL INFORMATION

You must specify:

- The symbolic device name, and, unless either SYSPCH or SYSLST is the symbolic device name used, the type of device that will write or punch the data set.
- The record format, record size, and, if the records are blocked, the block size, unless the defaults for stream-oriented output files are to apply. The defaults are applied as for record-oriented data transmission, described under "Record Format, BLKSIZE, and RECSIZE Defaults" in Chapter 7.

If you want to take advantage of the DOS device-independent data transmission facilities, files associated with the symbolic devices SYSPCH and SYSLST must have F-format records with a record length of 80 bytes for SYSPCH and 121 bytes for SYSLST. The device-independence facilities permit the assignment of the device address of a device other than a card punch or printer to these symbolic device names.

Dependent upon the level of your operating system, a program that uses stream-oriented transmission to create a CONSECUTIVE data set on a labeled magnetic tape must be link-edited with a LBLTYP statement present so that space is reserved for processing the tape label. When the program is executed, a TLBL job control statement must be given to identify the tape volume and the data set.

If the program uses stream-oriented transmission to create a CONSECUTIVE data set on a direct access volume, a DLBL and one or more EXTENT statements must be given to identify the direct access volume(s) and extent(s) for the data set.

Figure 29 summarizes the data set information that must be given for a data set created by stream-oriented transmission.

Always Required For:	Information Required	Where Specified
Any input/output using stream-oriented transmission	Type of device, unless device independent Symbolic device name Record format	File declaration in source program: see <u>OS and DOS PL/I Language Reference Manual</u>
Non-standard device assignment	Device assignment	ASSGN statement
Data set on magnetic tape with standard labels	Identification Storage for label processing	TLBL statement LBLTYP[1] statement
Data set on diskette or direct access volume	Identification and extent information	DLBL and EXTENT[2] statements

[1]The LBLTYP statement is not always necessary when operating under VSE/Advanced Functions.
[2]The EXTENT statement is not always necessary when using VSE/VSAM Space Management for SAM (see Chapter 7).

Figure 29. Data Set Information for Stream-Oriented Transmission

EXAMPLE

Figure 30 on page 111 illustrates the use of edit-directed stream-oriented transmission to create a data set on a disk storage volume.

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 data set associated with the PL/I file WORK has variable-length blocked (VB-format) records. Note that the PUT statement that transmits data to the file WORK has a SKIP option. The effect of the SKIP option is to create a new variable-length record for the data transmitted by each execution of this PUT statement.

An ASSGN statement is present to associate the symbolic device name SYS009 used in the program with the device address for the 3330 disk storage drive. A DLBL statement identifies the data set with the name PEOPLE and associates it with the PL/I file constant WORK. An EXTENT statement identifies the symbolic device name to be used in creating the data set, the storage volume that will hold the data set (DOS222), the type of storage extent represented by the EXTENT statement, the number of the track in which the data set is to commence, and the number of tracks to be reserved for the data set.

```
// JOB FIG0802
// OPTION LINK
// EXEC PLIOPT,SIZE=64K
   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;
/*
// EXEC LNKEDT
// ASSGN SYS009,3330,VOL=DOX222,SHR
// DLBL WORK,'PEOPLE',,SD
// EXTENT SYS009,DOS222,1,0,3458,19
// EXEC ,SIZE=64K
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 30. Creating a Data Set with Stream-Oriented Data Transmission

Figure 31 on page 112 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.

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.

ESSENTIAL INFORMATION

When accessing a data set using stream-oriented transmission, you must specify:

- The symbolic device name and, unless SYSIPT is the symbolic device name used, the device type of the device that will read the data set, in the MEDIUM option.
- The record format, record length, and, if records are blocked, the block size, unless the defaults for stream-oriented input files are to apply.

```
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 31. Writing Graphic Data to a Stream File

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 length 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 32 reads the data set created by the program in Figure 30 and uses the file SYSPRINT to list the data it contains. (SYSPRINT is discussed later in this chapter.) Each set of data is read, by the GET statement, into two variables: FREC, which always contains 45 characters; and VREC, into which is inserted the number of characters generated by the expression 7*NUM.

The symbolic device name SYS009 is assigned as in the previous example.

```
// JOB FIG0804
// OPTION LINK
// EXEC PLIOPT,SIZE=64K
  PEOPLE: PROCEDURE OPTIONS(MAIN);
          DECLARE WORK FILE STREAM INPUT
          ENV(VB RECSIZE(84) BLKSIZE(424) MEDIUM(SYS009,3330)),
          1 REC,
            2 FREC,
              3 NAME CHARACTER(20),
              3 NUM PIC '9',
              3 PAD1 CHAR(1),
              3 SERNO PIC '(6)9',
              3 PAD2 CHAR(1),
              3 PROF CHARACTER(16),
            2 VREC CHARACTER(35),
          IN CHAR(80) DEF REC;
          ON ENDFILE(WORK) GO TO FINISH;
          OPEN FILE(WORK),FILE(SYSPRINT);
  MORE:   GET FILE(WORK) EDIT(IN,VREC) (A(45),A(7*NUM));
          PUT FILE(SYSPRINT) SKIP EDIT (IN)(A);
          GOTO MORE;
  FINISH: CLOSE FILE(WORK),FILE(SYSPRINT);
          END PEOPLE;
/*
// EXEC LNKEDT
// ASSGN SYS009,3330,VOL=DOS222,SHR
// DLBL WORK,'PEOPLE'
// EXTENT SYS009,DOS222
// EXEC ,SIZE=64K
/&
```

Figure 32. Accessing a Data Set Using Stream-Oriented Transmission

A DLBL statement associates the PL/I file WORK with the data set PEOPLE, and an EXTENT statement identifies the symbolic device name and the disk storage volume DOS222 containing the data set. Data management will obtain the remainder of the information necessary to access the data set from the data set label that was written onto the volume when the data set was created.

PRINT FILES

Both the Disk 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 39 on page 131 through Figure 43 on page 132 in Chapter 9. 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.

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 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 that is compatible with the block size given, or with the defaults that are applied if these are not given.

Note that, if a PRINT file associated with a data set on a direct access volume is closed and reopened, any records written before the file is closed will be overwritten by any records transmitted after the file has been reopened.

EXAMPLE

Figure 33 illustrates the use of a PRINT file and the printing options of stream-oriented data transmission statements to

```
// JOB FIG0805
// OPTION LINK
// EXEC PLIOPT,SIZE=64K
 SINE: PROC OPTIONS(MAIN);
       DCL TABLE FILE PRINT ENV(VB MEDIUM(SYS006,3330)),
       HEADINGS CHAR(90) INIT('         0        6         12        18
   24        30       36        42        48        54'),
       TITLE CHAR(13) INIT('NATURAL SINES'),
       PGNO FIXED DEC(2) INIT(1),
       FINISH BIT(1) INIT('0'B),
       VALUES(0:359,0:9) FLOAT DEC(6);
       ON ENDPAGE(TABLE) BEGIN;
       PUT FILE(TABLE) EDIT('PAGE',PGNO)(LINE(55),COL(87),A,F(3));
       IF FINISH='0'B THEN DO;
              PGNO=PGNO+1
              PUT FILE(TABLE) EDIT(TITLE||' (CONT''D)',HEADINGS)
              (PAGE,A,SKIP(3),A);
              PUT FILE(TABLE) SKIP(2);
              END;
       END;
       DO I = 0 TO 359;
          DO J = 0 TO 9;
              VALUES(I,J) = SIND(I + J/10);3),10 F(9));
              END;
          END;
       OPEN FILE(TABLE) PAGESIZE(52) LINESIZE(93),FILE(SYSPRINT);
       PUT FILE(TABLE) EDIT(TITLE,HEADINGS)(PAGE,A,SKIP(3),A);
       DO I = 0 TO 71;
       PUT FILE(TABLE) SKIP(2);
              DO J = 0 TO 4;
              K=5*I+J;
              PUT FILE(TABLE)EDIT(K,VALUES(K,*))(F(3),10 F(9,4));
              END;
       END;
       FINISH='1'B;
       PUT FILE(TABLE)LINE(54);
       PUT EDIT('END OF SINE TABLE OUTPUT')(A)'
       CLOSE FILE(TABLE),FILE(SYSPRINT);
       END SINE;
/*
// EXED LNKEDT
// ASSGN SYS006,3330,VOL=DOS222,SHR
// DLBL TABLE,'SINES',,SD
// EXTENT SYS006,DOS222,1,0,3458,19
// EXEC ,SIZE=64K
/&
```

Figure 33. Creating a Data Set Using a PRINT File

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 example includes both an ASSGN statement for the symbolic device name SYS006 to associate it with the tape drive that is to be used, and a DLBL statement that associates the file TABLE with the data set SINES on volume number DOS222.

The program in Figure 44 on page 133 uses record-oriented data transmission to print the table created by the program in Figure 33.

TAB CONTROL TABLE

Data-directed and list-directed output to a PRINT file are automatically 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 size.
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 define the tab positions within the print line. The first position is numbered 1, and the highest position is numbered 255. The value of each tab should be greater than that of the tab preceding it in the table; otherwise, it is ignored. The first data field in the printed output begins at the next available tab position.

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 34 on page 117. 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 34. PL/I Structure PLITABS for Modifying the Preset Tab Settings

The second method is to create an assembler language control section called PLITABS and include it in the link-editing of the executable program.

SYSIN AND SYSPRINT FILES

PL/I includes a SYSIN file for input and a SYSPRINT file for output. If your program includes a GET statement that does not include the FILE or STRING option, the compiler inserts the name SYSIN; if it includes a PUT statement without the FILE or STRING option, the compiler inserts the name SYSPRINT.

If you do not declare SYSPRINT, the compiler will give the file the attribute PRINT in addition to the normal default attributes. Because SYSPRINT is a PRINT file, the compiler also supplies a default LINESIZE of 120 characters. Therefore, you need give only a minimum of information in the PL/I program. The complete statement for SYSPRINT will be assumed as follows:

```
DCL SYSPRINT FILE STREAM OUTPUT PRINT
             EXTERNAL ENV(F RECSIZE(121)
             MEDIUM (SYSLST));
```

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 transient library, and that any change you make in the format of the output from SYSPRINT will also apply to the format of execution-time diagnostic messages. If a line size less than 72 is used, any diagnostic messages will be transmitted to the operator console.

The compiler does not supply any attributes other than the defaults for the input file SYSIN; if you do not declare it, the compiler assumes the attributes:

```
DCL SYSIN FILE STREAM INPUT EXTERNAL
    ENV(RECSIZE(80) F MEDIUM(SYSIPT));
```

CHAPTER 9. USING CONSECUTIVE, INDEXED, AND REGIONAL DATA SETS

This chapter describes how to use CONSECUTIVE, INDEXED, and REGIONAL data sets using the SAM, QSAM, ISAM, and DAM access methods.

Figure 35 shows the facilities that are available with the various types of data sets that can be used with PL/I.

COMPARISON OF PL/I DATA SET TYPES							
	VSAM KSDS	VSAM ESDS	VSAM RRDS	INDEXED	CONSECUTIVE	REGIONAL (1)	REGIONAL (3)
SEQUENCE	Key Order	Entry Order	Num-bered	Key Order	Entry Order	By Region	By Region
DEVICES	DASD	DASD	DASD	DASD[1]	DASD,tape, card, etc.	DASD[2]	DASD
ACCESS 1=By key 2=Sequen-tial 3=Backward	123	123	123	12	2 3 tape only	12	12
Alternate index Access as above	123	123	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
DELETION 1=Space reusable 2=Space not reusable	Yes,1	No	Yes,1	No	No	No	No

[1] Limited to 2311, 2314, 3330-1, and 3340 devices.
[2] Not FBA devices.

Figure 35. A Comparison of Data Set Types Available to PL/I Record I/O

CREATING AND ACCESSING DATA SETS FOR RECORD-ORIENTED TRANSMISSION

To create or access a data set for record-oriented transmission, you must give the operating system information in your PL/I program, and, in certain cases, in TLBL or DLBL and EXTENT and other job control statements. These requirements are summarized in Figure 36 on page 119.

In record-oriented transmission, data is transmitted to and from auxiliary storage exactly as it appears in the program variables; no data conversion takes place for data sets other than ASCII data sets or for data that requires conversion when written onto or read from 7-track magnetic tape. A record in a data set corresponds to a variable in the program.

The following sections describe the information that you must supply, and discuss some of the information you may supply, when processing CONSECUTIVE, INDEXED, and REGIONAL data sets.

Always Required for:	Information	Where Specified
Any input/output using record-oriented transmission	Type of device - unless device independent Symbolic device name Record format	File declaration in source program: see the ENVIRONMENT attribute MEDIUM option in Chapter 7.
Nonstandard device assignment	Device assignment	ASSGN statement
Data set on magnetic tape with standard labels	Identification Storage for label processing	TLBL statement LBLTYP statement[1]
Data set on direct-access volume	Identification and extent information REGIONAL(1),REGIONAL(3), or INDEXED data set, storage for label processing	DLBL and EXTENT[2] statements LBLTYP[1]

[1] Not required for VSE/Advanced Functions.

[2] The EXTENT statement is not always necessary when using VSE/VSAM Space Management for SAM (see Chapter 7).

Figure 36. Record-Oriented Transmission Data Set Information

CONSECUTIVE DATA SETS

This section describes CONSECUTIVE data set organization, the data transmission statements used with CONSECUTIVE data sets, and the ENVIRONMENT options that define CONSECUTIVE data sets. It then describes how to create, access, and update CONSECUTIVE data sets.

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 37 on page 120 lists the data transmission statements and options that you can use to create and access a CONSECUTIVE data set.

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(option-list);
```

The file attributes are described in the _OS and DOS PL/I Language Reference Manual_. Default file attributes are shown in Figure 25 on page 95.

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	READ FILE(file-reference) INTO(reference);	
	READ FILE(file-reference) SET(pointer-reference);	
	READ FILE(file-reference) IGNORE(expression);	
SEQUENTIAL INPUT UNBUFFERED	READ FILE(file-reference) INTO(reference);	EVENT(event-reference)
	READ FILE(file-reference) IGNORE(expression);	EVENT(event-reference)
SEQUENTIAL UPDATE BUFFERED	READ FILE(file-reference) INTO(reference);	
	READ FILE(file-reference) SET(pointer-reference);	
	READ FILE(file-reference) IGNORE(expression);	
	REWRITE FILE(file-reference);	FROM(reference)
SEQUENTIAL UPDATE UNBUFFERED	READ FILE(file-reference) INTO(reference);	EVENT(event-reference)
	READ FILE(file-reference) IGNORE(expression);	EVENT(event-reference)
	REWRITE FILE(file-reference) FROM(reference);	EVENT(event-reference)

[1]The complete file declaration would include the attributes FILE, RECORD, and ENVIRONMENT.

[2]The statement READ FILE (file-reference); is equivalent to: READ FILE(file-reference) IGNORE (1);

Figure 37. CONSECUTIVE Data Set Statements and Options

ENVIRONMENT OPTIONS FOR CONSECUTIVE DATA SETS

The following options apply only to CONSECUTIVE data sets and are described in this chapter:

ASCII	LEAVE | UNLOAD
ASSOCIATE	NOFEED
BUFOFF	NOLABEL
CMDCHN	NOTAPEMK
COLBIN	OMR
CONSECUTIVE	RCE
CTLASA | CTL360	STACKER
FILESEC	VOLSEQ
FUNCTION	WRTPROT

The following options apply to CONSECUTIVE as well as one or more other data set organization:

BLKSIZE	MEDIUM
BUFFERS	RECSIZE
COBOL	SCALARVARYING
F|FB|V|VB|D|DB|U	VERIFY

These options are described in Chapter 7. Figure 26 on page 96 summarizes the ENVIRONMENT 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-string 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.

Because 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.

ASSOCIATE Option

The ASSOCIATE option of the ENVIRONMENT attribute enables multiple operations (for example, read and then punch) to be performed on an IBM 3525 Card Punch. A file is declared for each operation that is required, and the files are "associated" with each other by means of the ASSOCIATE option.

Syntax

```
ASSOCIATE(filename)
```

- The filename must be a file constant declared in the same external procedure.
- The device type specified in the MEDIUM option must be 3525 or 3525T.

- The logical unit specified in the MEDIUM option must not be SYSLST, SYSIPT, or SYSPCH.
- The file must have the RECORD attribute.
- BUFFERS(1) must be specified explicitly or by default for files that are used to read or to punch cards.
- The FUNCTION option must be specified with a second operand that corresponds to the set of associated files.

Files must be associated in the order "read - punch - write - read." The following example shows the declarations for three files that are used to read a card, punch it, and then print on it.

```
DCL READ FILE RECORD INPUT
 ENV(..FUNCTION(R,RPW),ASSOCIATE(PUNCH)..),

PUNCH FILE RECORD OUTPUT
 ENV(..FUNCTION(P,RPW),ASSOCIATE(PRINT)..),

PRINT FILE RECORD OUTPUT
 ENV(..FUNCTION(W,RPW),ASSOCIATE(READ)..);
```

Input/output operations on sets of associated files must be performed in the order in which the files are associated. For the files given in the example above, the correct sequence is:

```
READ FILE (READ) . . . . . ;
WRITE FILE (PUNCH) . . . . ;
WRITE FILE (PRINT) . . . . ;
```

You must ensure that the correct sequence is followed. If, however, the sequence contains a print operation, any number of CONSECUTIVE print operations may be performed, from zero (print operation omitted) up to the maximum for the device (2 or 25).

The first file in the input/output sequence may be opened implicitly. The remaining files must be opened explicitly before an operation is performed on any of the associated files.

You must ensure that none of the associated files is closed before all the I/O operations on the set of associated files are complete. For this reason, a multiple close statement should be used to close a set of associated files if locate-mode I/O is being used.

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. Therefore, the only permissible buffer offset specification on output is 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.

If ASCII is not specified but one of BUFOFF, D, or DB is specified, then ASCII is assumed.

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.

CMDCHN Option

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

Syntax

```
CMDCHN(n)
```

If the CMDCHN option is specified, the file must be explicitly opened. If the CMDCHN option is not specified, CMDCHN(1) is assumed.

COLBIN Option

The COLBIN option specifies that cards processed by an IBM 3505 or 3525 Card Reader hold data in column binary form.

Syntax

```
COLBIN
```

- The file must have the RECORD attribute.
- The device type in the MEDIUM option must be 3505, 3525, or 3525T.
- The FUNCTION option must not specify punch interpret (FUNCTION(P,I)) or write (FUNCTION(W)).

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 data set organization.

CTLASA and CTL360 Options

The printer and punch control options, CTLASA and CTL360, apply only to OUTPUT files associated with CONSECUTIVE data sets. They specify that the first character of a record is to be interpreted as a control character.

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 later in this chapter.

FILESEC Option

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

Syntax

```
FILESEC
```

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

FUNCTION Option

The FUNCTION (device function) option specifies the operations that are to be performed by an IBM 3525 Card Punch, IBM 2560 Card Read Punch, or IBM 5425 Card Read Punch. The FUNCTION option for the IBM 3525 has the format:

```
Syntax

                  ┌,I  ┐
             {R}  │,RP │
FUNCTION (   {P}  │,RW │   )
             {W}  │,PW │
                  └,RPW┘
```

The FUNCTION option for the IBM 2560 and IBM 5425 has the format:

```
Syntax

             {R}
FUNCTION (   {P}   [1|2] )
             {W}
```

The letters R, P, and W specify the operation that is to be performed by the file, as follows:

R The file is used to read cards. The file must have the INPUT attribute.

P The file is used to punch cards. The file must have the OUTPUT attribute, and must not have the PRINT attribute.

W The file is used to print on cards. The file must have the OUTPUT attribute; for the IBM 3525 only, the file may have neither the PRINT attribute nor the CTLASA or CTL360 option, since no printer control characters are available on a 2560 or 5425.

The digits 1 and 2, applicable only to IBM 2560 and IBM 5425, indicate the input stacker selection for the file.

The second operand of the FUNCTION option, applicable only to the IBM 3525, specifies that the device is being used for multiple operations on the same card. It can have the following values:

I The file is used to punch and interpret files. In this case only, the first operand may be omitted; if it is specified, it must be P.

RP The file is one of two associated files that are used to read and then punch cards.

RW The file is one of two associated files that are used to read and then print on cards.

PW The file is one of two associated files that are used to punch and then print on cards.

RPW The file is one of three associated files that are used to read, punch, and then print on cards.

The following rules apply to the FUNCTION option for the IBM 3525:

- The device type specified in the MEDIUM option must be 3525 or 3525T.
- If the second operand of the FUNCTION option is I, the first operand, if specified, must be P.
- The COLBIN option must not be specified with FUNCTION(P,I).
- The ASSOCIATE option must be specified, unless FUNCTION has a single operand or is FUNCTION(P,I).
- If FUNCTION(P,I) is specified, the output records must be F format.

If the FUNCTION option is omitted from the declaration of a file associated with a 3525, the following defaults are applied:

INPUT files	FUNCTION(R)
OUTPUT non-PRINT files	FUNCTION(P)
OUTPUT PRINT files	FUNCTION(W)

The defaults for the IBM 2560 and IBM 5425 are:

INPUT files	FUNCTION(R1)
OUTPUT files	FUNCTION(P1)

(If only the stacker selection digit is omitted, the default is stacker 1.)

LEAVE and UNLOAD Options

The LEAVE and UNLOAD 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.

Syntax

LEAVE | UNLOAD

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).

The UNLOAD option is used to specify that the tape is to be rewound and unloaded when the file is closed, or (for input files) when a tape mark is read.

NOFEED Option

The NOFEED option is permitted only with the IBM 3540 Diskette. It is used to specify that the diskette is to remain in position at the end of a job step, so that it may be accessed later without operator intervention.

Syntax

NOFEED

NOLABEL Option

The NOLABEL option is used to specify that no file labels are to be processed for a magnetic tape file.

Syntax

```
NOLABEL
```

If the NOLABEL option is specified for output files, a tape mark is written as the first record on the tape unless, in addition to NOLABEL, the NOTAPEMK option is specified in the ENVIRONMENT attribute. Nonstandard labels and additional user labels are not processed.

NOTAPEMK Option

The NOTAPEMK option for tape files enables you to prevent a leading tapemark from being written ahead of the data records on unlabeled tape files. The resulting data set cannot be read backward, unless it is an ASCII data set.

Syntax

```
NOTAPEMK
```

NOTAPEMK may be used for tape OUTPUT files with NOLABEL specified. This option is not allowed for UNBUFFERED files.

OMR Option

The OMR (optical mark read) option specifies the optical reading of marks in a standard 80-column card by a 3505. If this option is used, the first card in the deck of cards to be read by the program must be an OMR format descriptor card.

Syntax

```
OMR
```

- The file must have the RECORD attribute.
- BUFFERS(1) must be specified explicitly or by default.
- The device type in the MEDIUM option must be 3505.
- The Read Column Eliminate (RCE) option must not be specified.

If a block size (BLKSIZE) of less than 80 (EBCDIC) or 160 (column binary) is specified, the block size is changed to 80 or to 160 as appropriate.

RCE Option

The RCE (read column eliminate) option specifies the selective reading of card columns by an IBM 3505 Card Reader or a 3525 Card Punch. If this option is used, the first card in the deck of cards to be read by the program must be an RCE format descriptor card. The format of the format descriptor card is described under "Features of the IBM 3504, 3505, and 3525" on page 82 in Chapter 7.

Syntax

```
RCE
```

- The device type specified in the MEDIUM option must be 3505, 3525, or 3525T.
- The logical unit specified in the MEDIUM option must not be SYSIPT.
- The Optical Mark Read (OMR) option must not be specified.

STACKER Option

The STACKER option specifies into which stacker of an IBM 3505 Card Reader or IBM 3505 Card Punch cards are to be directed.

Syntax

```
STACKER(n)
```

where n is a constant or a FIXED BINARY STATIC variable of precision (31,0).

- The device type specified in the MEDIUM option must be 3505, 3525, or 3525T.
- The FUNCTION option must not specify write (FUNCTION(W)).
- n must be either 1 or 2.
- The stacker that is selected depends on the value of n when the file is opened (either explicitly or implicitly). It cannot be changed unless the file is closed and then reopened.

If the STACKER option is not specified, STACKER(1) is assumed.

VOLSEQ Option

The VOLSEQ (volume sequence) option is permitted only with IBM 3540 Diskette input files. For multivolume data sets, it is used to specify that sequence checking on the volume serial numbers must be performed.

Syntax

```
VOLSEQ
```

WRTPROT Option

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

Syntax

```
WRTPROT
```

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 the LOCATE statement may be used to write records. Figure 38 on page 129 shows the statements and options permitted for creating a CONSECUTIVE data set.

Essential Information

When you create a CONSECUTIVE data set using record-oriented transmission, you must specify:

- The symbolic device name, and, unless either SYSPCH or SYSLST is the symbolic device name used, the type of device that will write or punch the data set.
- The record format, record size, and, if the records are blocked, the block size. No defaults are permitted.

```
// JOB FIG0904
// OPTION LINK
// EXEC PLIOPT,SIZE=64K
 MERGE: PROCEDURE OPTIONS(MAIN);
        DCL IN1 FILE RECORD SEQUENTIAL
            ENV(F RECSIZE(15) MEDIUM(SYS006,3330)),
            IN2 FILE RECORD SEQUENTIAL ENV(F RECSIZE(15) MEDIUM(SYS007,3330)),
            OUT FILE RECORD SEQUENTIAL ENV(MEDIUM(SYS008,3330)
                                           F RECSIZE(15))OUTPUT,
            (ITEM1 BASED(A),ITEM2 BASED(B)) CHAR(15);
          /* START OF ENDFILE ON-UNITS */
         ON ENDFILE(IN1) BEGIN;
           ON ENDFILE(IN2) GO TO FINISH;
 NEXT2:    WRITE FILE(OUT) FROM(ITEM2);
           PUT SKIP LIST (ITEM2);
           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 SKIP LIST (ITEM1);
           READ FILE(IN1) SET(A);
           GO TO NEXT1;
           END;
           /* END OF ENDFILE ON-UNITS */
         OPEN FILE(IN1),FILE(OUT),FILE(IN2),FILE(SYSPRINT);
         READ FILE(IN1) SET(A);
         READ FILE(IN2) SET(B);
 NEXT:   IF ITEM1 > ITEM2 THEN DO;
           PUT SKIP LIST(ITEM2);
           WRITE FILE(OUT) FROM(ITEM2);
           READ FILE(IN2) SET(B);
           GO TO NEXT;
           END;
         ELSE DO;
           PUT SKIP LIST(ITEM1);
           WRITE FILE(OUT) FROM(ITEM1);
         READ FILE(IN1) SET(A);
         GO TO NEXT;
         END;
 FINISH: CLOSE FILE(IN1),FILE(IN2),FILE(OUT),FILE(SYSPRINT);
         END MERGE;
/*
// EXEC LNKEDT
// ASSGN SYS006,3330,VOL=DOS222,SHR
// ASSGN SYS007,3330,VOL=DOS222,SHR
// ASSGN SYS008,3330,VOL=DOS222,SHR
// DLBL IN1,'DS1'
// EXTENT ,DOS222
// DLBL IN2,'DS2'
// EXTENT ,DOS222
// DLBL OUT,'DS3'
// EXTENT ,DOS222,1,0,3458,2
// EXEC ,SIZE=64K
/&
```

Figure 38. Creating and Accessing a CONSECUTIVE Data Set

ACCESSING A CONSECUTIVE DATA SET

To access an existing data set on a labeled magnetic tape or direct access device, you must identify it to the operating system in a TLBL statement or DLBL and EXTENT statements. Figure 38 shows the statements and options permitted for accessing a CONSECUTIVE data set.

Essential Information

When accessing a CONSECUTIVE data set using record-oriented transmission, you must specify:

- The symbolic device name, and, unless SYSIPT is the symbolic device name used, the type of device that will read the data set.
- The record format, record size, and, if the records are blocked, the block size. No defaults are permitted.

Record Format

The record format information must be compatible with the actual structure of the data set. For example, if you create a data set with FB-format records, a record size of 600 bytes, and a block size of 3600 bytes, you must either access them with a file declared with the ENVIRONMENT options FB, RECSIZE(600) BLKSIZE(3600), or you can access the records as if they were undefined-length records with a maximum block size of 3600 bytes; but if you specify undefined-length records with a maximum block size of 3500 bytes, the blocks will be truncated.

EXAMPLE OF CONSECUTIVE DATA SETS

Creating and accessing CONSECUTIVE data sets is illustrated in the program shown in Figure 38. The program merges the contents of two existing data sets, DS1 and DS2, and writes them onto a new data set, 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. Each of the data sets is identified and associated with the appropriate PL/I file by a DLBL statement. ASSGN statements to establish the symbolic device names used by each of the files are also included.

PUNCHING CARDS AND PRINTING

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 in your PL/I program (CTL360 or CTLASA ENVIRONMENT option). 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, the job will be terminated.

The American National Standard control characters, listed in Figure 39, cause the specified action to occur before the associated record is printed or punched.

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 39. American National Standard Print and Card Punch Control Characters (CTLASA)

Code Byte	Action
00000001	Select stacker 1
01000001	Select stacker 2
10000001	Select stacker 3

Figure 40. 2540 Card Read Punch Control Characters (CTL360)

Print, Then Act Code Byte	Action	Act Immediately (no printing) 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 chnl 1	10001011
10010001	Skip to chnl 2	10010011
10011001	Skip to chnl 3	10011011
10100001	Skip to chnl 4	10100011
10101001	Skip to chnl 5	10101011
10110001	Skip to chnl 6	10110011
10111001	Skip to chnl 7	10111011
11000001	Skip to chnl 8	11000011
11001001	Skip to chnl 9	11001011
11010001	Skip to chnl 10	11010011
11011001	Skip to chnl 11	11011011
11100001	Skip to chnl 12	11100011

Figure 41. IBM Machine Code Print 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 42. 3525 Card Printer Control Characters (CTLASA)

The machine code control characters differ according to the type of device. The IBM machine code control characters for the IBM 2540 Card Read Punch are listed in Figure 40 on page 131, and Figure 41 gives those for printers. Control codes for the IBM 3525 Card Printer are given in Figure 42 and Figure 43.

Code Byte	Action
00001101	Print on line 1
00010101	Print on line 2
00011101	Print on lien 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 43. 3525 Card Printer Control Characters (CTL360)

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 44 uses record-oriented transmission to read and print the contents of the data set SINES, which was created by the PRINT file in Figure 33 on page 115. The output file PRINTER is declared with the option CTLASA, ensuring that the first byte of each record will be interpreted as an ANS printer control code. The example requires an ASSGN statement for the symbolic device name SYS006, and a DLBL statement to associate the file TABLE with the data set SINES on the volume number DOS222.

```
// JOB FIG0910
// OPTION LINK
// EXEC PLIOPT,SIZE=64K
 PRT:    PROC OPTIONS(MAIN);
         DCL TABLE FILE RECORD INPUT SEQUENTIAL
             ENV(VB RECSIZE(98) BLKSIZE(494) MEDIUM(SYS006,3330)),
             PRINTER FILE RECORD OUTPUT SEQUENTIAL
                     ENV(MEDIUM(SYSLST,1403)
                     V RECSIZE(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;
/*
// EXEC LNKEDT
// ASSGN SYS006,3330,VOL=DOS222,SHR
// DLBL TABLE,'SINES'
// EXTENT SYS006,DOS222
// EXEC ,SIZE=64K
/&
```

Figure 44. 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 this chapter.

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 Chapter 7. 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.

- When a read or punch associated file is opened, the value of the BUFFERS option will be set to 1.

- 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 BUFFERED 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, data transmission statements, and the ENVIRONMENT options that define INDEXED data sets. It then describes how to create, access, and reorganize INDEXED data sets. See Figure 45 on page 135.

INDEXED ORGANIZATION

A data set with INDEXED organization must be on a direct access device. Its records, which can be either F-format blocked or unblocked records, 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, 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

File Declaration[1]	**Valid Statements**[2]**, 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)
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(file-reference) FROM(reference) KEYFROM(expression);	EVENT(event-reference)

[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);

<u>Note</u>: The attribute UNBUFFERED is ignored and BUFFERED is the default for SEQUENTIAL files.

Figure 45. INDEXED Data Set Statements and Options

external storage space than a CONSECUTIVE data set, and all volumes of a multivolume data set must be mounted, even for sequential processing.

Figure 45 on page 135 lists the data transmission statements and options that can be used to create and access an INDEXED data set.

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. See Figure 46.

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 that the operating system develop a _master index_ for a specified number of cylinders.

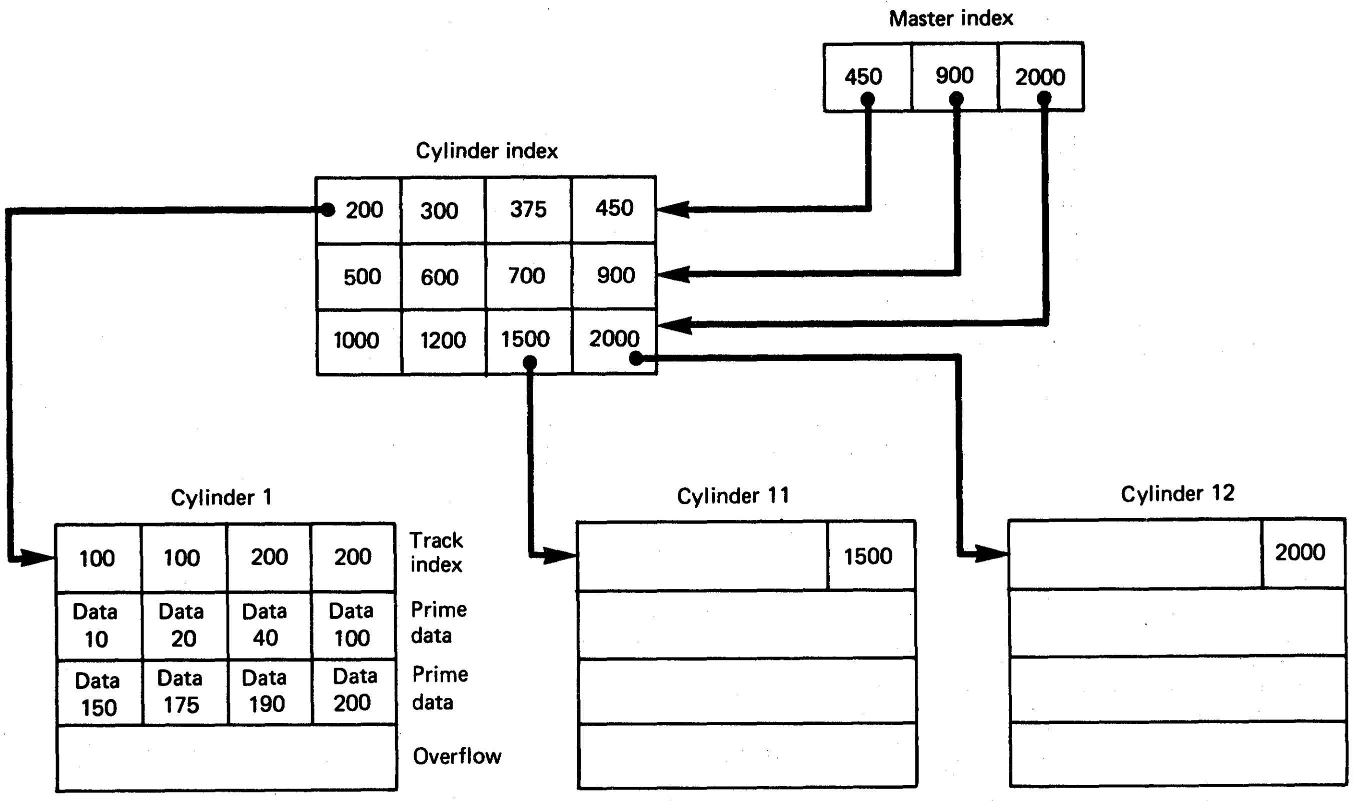


Figure 46. Index Structure of an INDEXED Data Set

You can have up to three levels of master index; Figure 46 illustrates the index structure. The part of the data set that contains the cylinder and master indexes is termed the index area.

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 47 on page 138. 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.

Keys

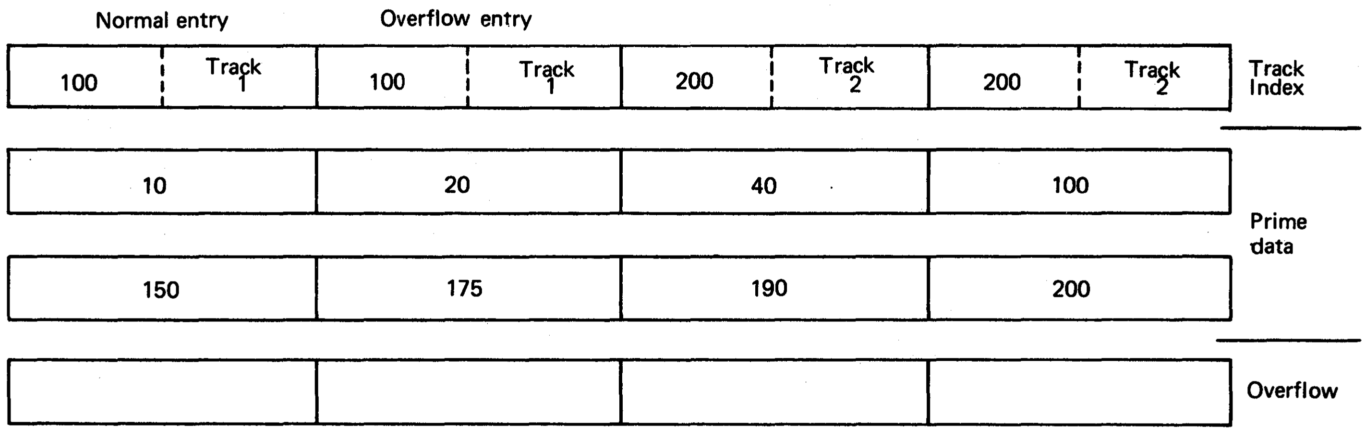
There are two kinds of keys—recorded keys and source keys. A recorded key is a character string that actually appears with each record in the data set to identify that record; its length cannot exceed 255 characters and all keys in a data set must have the same length. The recorded keys in an INDEXED data set may be separate from, or embedded within, the logical records. A source key is the character-string 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.

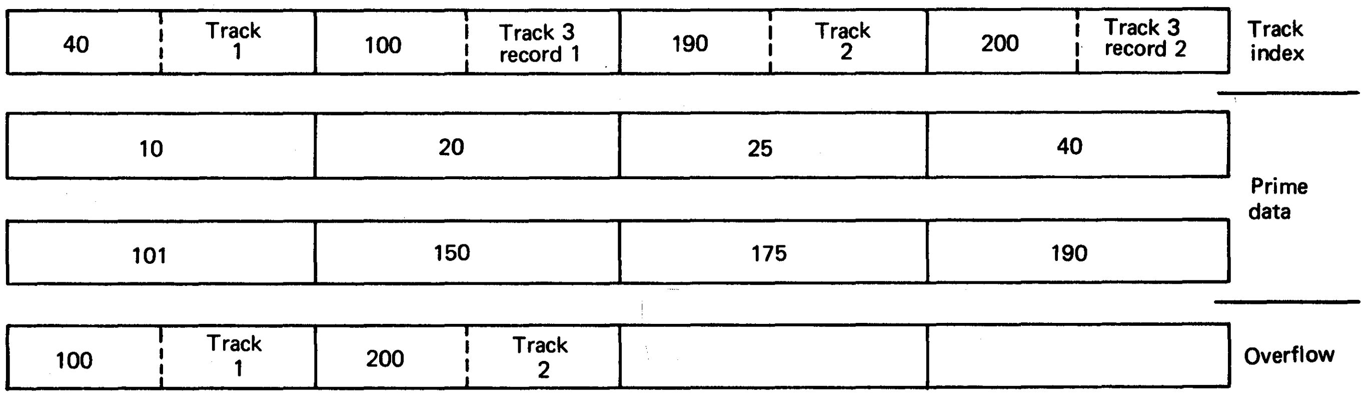
Embedded Keys

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.

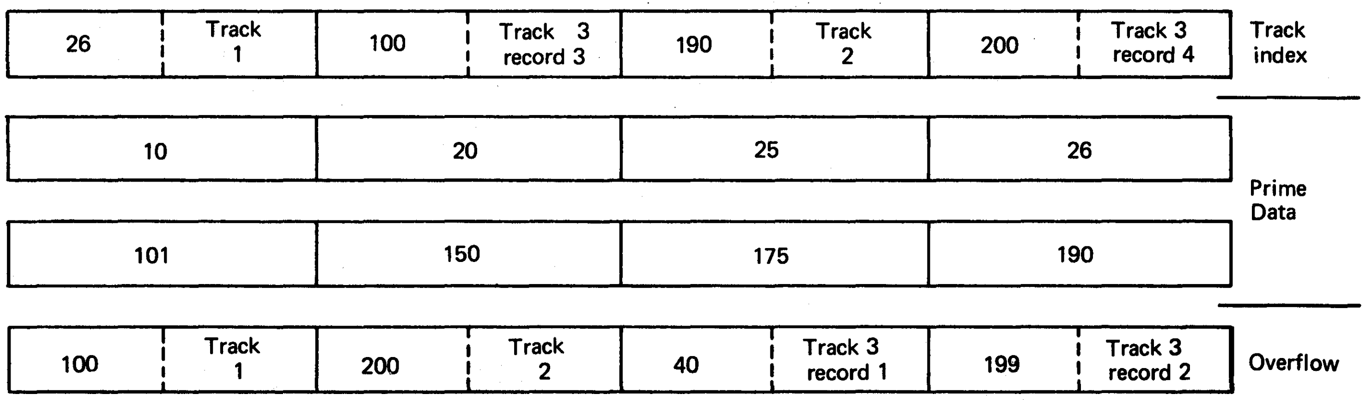
During the execution of a WRITE statement that adds a record to a data set with embedded keys, the value of the expression in the KEYFROM option is assigned to the embedded key position in the record variable. Note that a record variable can be



Initial format of an INDEXED data set



INDEXED data set after addition of records 26 and 101



INDEXED data set after addition of records 26 and 199

Figure 47. Adding Records to an INDEXED Data Set

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.

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(option-list);
```

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
             ENVIRONMENT(option-list);
```

The file attributes are described in the OS and DOS PL/I Language Reference Manual. Default attributes are shown in Figure 25 on page 95.

ENVIRONMENT OPTIONS FOR INDEXED DATA SETS

The following options apply only to INDEXED data sets and are described in this chapter.

ADDBUFF	INDEXMULTIPLE
HIGHINDEX	KEYLOC
INDEXAREA	NOWRITE
INDEXED	OFLTRACKS

The following options apply to INDEXED as well as one or more other data set organization:

BLKSIZE	KEYLENGTH
BUFFERS	MEDIUM
COBOL	RECSIZE
EXTENTNUMBER	SCALARVARYING
F|FB	VERIFY

These options are described in Chapter 7. Figure 26 on page 96 summarizes the ENVIRONMENT options.

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.

Syntax

ADDBUFF(n)

The ADDBUFF option need not be specified for DIRECT INDEXED files with V-format records, as the workspace is automatically allocated for such files.

HIGHINDEX Option

The HIGHINDEX option is used for INDEXED data sets to specify the type of device (2311, 2314, 3330, or 3340) on which the high-level index or indexes reside(s) if the device type differs from the one specified in the MEDIUM option.

Syntax

```
HIGHINDEX(device-type)
```

Note: When 3330 is specified, it is assumed to mean 3330 model 1; the 3330-11 device is not supported for INDEXED organization.

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 specification 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.

INDEXED Option

The INDEXED option defines a file with INDEXED organization (which is described above).

Syntax

```
INDEXED
```

INDEXMULTIPLE Option

The INDEXMULTIPLE option is used for INDEXED data sets to specify that a master index will be, or has been, built for this file.

Syntax

```
INDEXMULTIPLE
```

KEYLOC Option

The KEYLOC (key location) option can be used with INDEXED data sets, when the data set is created, to specify the absolute position of an embedded key from the start of the data in a record.

Syntax

```
KEYLOC(n)
```

The position given (n) must be within the limits:

1 <= n <= record length - 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, the KEYLOC(n) option should be specified.

The equivalent KEYLOC value for a particular byte is affected by the following:

- The KEYLOC byte count starts at 1
- The record format

For example, if the embedded key begins at the 10th byte of a record variable, then the specifications are:

Fixed length: KEYLOC(10)

Variable-length: KEYLOC(10)

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(1) is specified, it must be specified for every file that accesses the data set. The effect of the use of the KEYLOC option is shown in Figure 48.

KEYLOC(n)	RECORD VARIABLE	DATA SET UNBLOCKED RECORDS	DATA SET BLOCKED RECORDS
n > 1	Key	Key	Key
n = 1	Key	Key[1]	Key
n = 0 or not specified	No key	No key	Key[2]
	Key	Key	Key

[1] In this instance, the key is not recognized by data management.
[2] Each logical record in the block has a key.

Figure 48. Effect of KEYLOC Values on Establishing Embedded Keys

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.

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
```

OFLTRACKS Option

The OFLTRACKS option is used for OUTPUT files and INDEXED organization to specify the number of tracks to be reserved on each cylinder for adding records.

Syntax

```
OFLTRACKS(n)
```

The number specified (n) must be within the following limits:

0 <= n <= 8 for data sets on 2311

0 <= n <= 17 for data sets on 3330-1

0 <= n <= 18 for all other devices

For INPUT files and UPDATE files, when no records are to be added, this option is ignored.

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. Figure 45 on page 135 shows the statements and options permitted for creating an INDEXED data set.

A single EXTENT statement cannot define the whole of an INDEXED data set. You must use EXTENT statements to define the following separately:

- The storage area for the master index, if used.
- The storage area for the cylinder index. This is always required.
- The storage area for each prime data area in the data set. At least one prime data area is required.
- The storage area for an independent overflow area, if used.

Essential Information

The DLBL statement should contain the data set organization code ISC (INDEXED Sequential Creation) and precede the associated EXTENT statements. The space for an INDEXED data set must be allocated as one or more complete cylinders. If the data set spans more than one direct access volume, it must continue from the last track of one volume to the first track of the second cylinder (cylinder 1) of the following volume. When the program is link-edited, it will be necessary to include a LBLTYP statement in the job, unless you are operating under VSE/Advanced Functions.

If a master index is required, its extent must be adjacent to that of the cylinder index. When creating an INDEXED data set, you must specify:

- The device type and symbolic device name of the direct access device that will contain the new data set, in the MEDIUM option.
- The record format F for fixed-length unblocked records or FB for fixed-length blocked records, the record size, and, if blocked records are used, the block size.
- The data set organization INDEXED.
- The KEYLENGTH option to specify the length of the recorded keys.
- The EXTENTNUMBER option to specify the maximum number of extents used for the data set. (The number should exclude the EXTENT statements supplied for a master index, if used.) A LBLTYP statement will also be necessary, unless you are operating under VSE/Advanced Functions, and should specify the full number of EXTENT statements.
- The OFLTRACKS option to indicate the number of overflow tracks per cylinder in the prime data area.

You can specify the VERIFY option to ensure that each record is correctly written onto the direct access volume.

If a master index is to be created, specify the INDEXMULTIPLE option. If such a master index and the cylinder index are to be held on a device separate from the remainder of the data set, specify the HIGHINDEX option to indicate the type of device used.

Master Index

The use of a master index is not recommended unless the cylinder index occupies more than three tracks. The EXTENT statement for the master index must be the first to follow the DLBL statement for the INDEXED data set. The EXTENT statement must have the type code 4 and the sequence number 0. It must also occupy storage preceding and immediately adjacent to the storage allocated for the cylinder index.

Cylinder Index

The cylinder index is always required. The EXTENT statement for the cylinder index must follow an EXTENT statement for a master index, if present, and have the type code 4 and the sequence number 1. The storage for the cylinder index must immediately follow that of a master index, if used.

Prime Data Area

An EXTENT statement must be supplied to define each prime data area in an INDEXED data set. One prime data area only is permitted on a single direct access volume. For a data set that spans two or more direct access volumes, a separate EXTENT statement must be supplied for one prime data area on each volume. A prime data area must start and end on a full cylinder boundary.

The EXTENT statement must have the type code 1. The sequence numbers for prime data area EXTENT statements start at 2.

Overflow Areas

If records are to be added to the data set after its initial creation, it may be necessary to provide storage for records that cannot be accommodated when a particular track is full.

To specify a cylinder overflow area, the overflow area that is a part of the prime data, use the OFLTRACKS option for the associated PL/I file. OFLTRACKS causes the number of tracks specified to be reserved in each cylinder of the prime data area as the overflow area for the remaining tracks on the cylinder. OFLTRACKS need only be specified for an output file or an update file to which records are to be added. The value specified must not vary for a particular data set.

To specify an independent overflow area, provide an EXTENT statement with the type code 2. Only one independent overflow area is permitted for a data set. The independent overflow area cannot span separate direct access volumes.

The use of either type of overflow area has the advantage of reducing the amount of unused space in the prime data area, but entails an increased search time for overflow records.

It is good practice to request an overflow area large enough to contain a reasonable number of additional records.

The use of an independent overflow area on a volume separate from that which contains the prime data area will help to increase the speed of direct access to the records in the data set by reducing the number of access mechanism movements required.

If all the tracks in the prime data area are not filled during creation, you cannot use the unused portion of the prime data area for overflow records from existing full tracks when records are subsequently added during direct access (although the unfilled portion of the last track used and any unused tracks in the prime data area can be filled with records that are in the correct key sequence to be added to the end of the data set). You can reserve space for later use within the part of the prime data area that already contains records by writing dummy records during creation.

Record Format and Keys

An INDEXED data set can contain only fixed-length records, blocked or unblocked. You must always specify the record format in the PL/I program.

If the records are blocked, each record must include the associated key (that is, the key must be embedded in the record). An embedded key can appear in any position within the record, and you must use the KEYLOC option of the ENVIRONMENT attribute to indicate its position.

Unblocked records can have embedded keys or the keys can be separated from the records. If the keys are not embedded, either specify KEYLOC(0) or omit the KEYLOC option. Figure 49 on page 145 shows the relationship between record size and block size and Figure 50 on page 146 illustrates the record formats for an INDEXED data set.

Creating Dummy Records and Deleting Records

You cannot change the specification of an INDEXED data set after you have created it. Therefore, you must foresee your future needs where the size and location of the index, prime, and overflow areas are concerned. The "deletion" of unwanted records and the recognition of "deleted" records is your responsibility. A record can be deleted by converting it into a null or dummy record. The program should test each record retrieved to determine whether it contains valid data, or whether it is a null or dummy record.

	KEYLOC OPTION	RECSIZE OPTION	BLKSIZE OPTION	KEYLENGTH OPTION
BLOCKED RECORDS	Always required; default is KEYLOC(1)	R	R*B	Always required
UNBLOCKED RECORDS	Required if key is embedded; otherwise, omit or specify KEYLOC(0)	R	R	

R = Size of record B = Blocking factor

Example: Blocked records with embedded key of 20 bytes starting at byte 10 of each record. Records are 120 bytes in length and blocked in groups of 10:

```
...ENVIRONMENT(INDEXED KEYLOC(10) KEYLENGTH(20)
               FB RECSIZE(120) BLKSIZE(1200)
                 EXTENT NUMBER(3) OFLTRACKS(2)...)....
```

Figure 49. Record Format Information for an INDEXED Data Set

ACCESSING AN INDEXED DATA SET

After 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 45 on page 135 shows the statements and options permitted 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 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.

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.

The EVENT option is not supported for SEQUENTIAL access of INDEXED data sets.

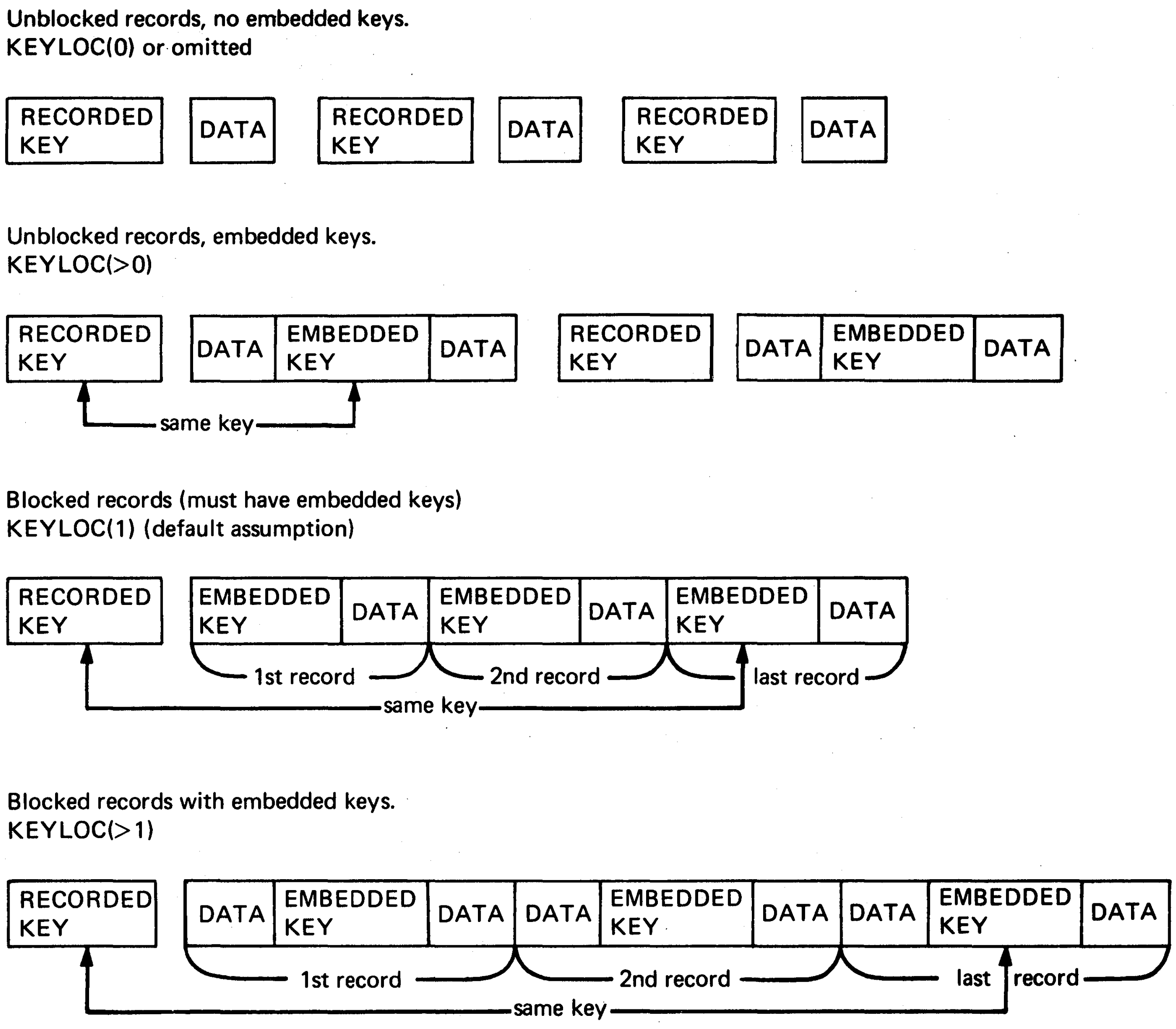


Figure 50. Record Formats in an INDEXED Data Set

INDEXED KEYED files opened for SEQUENTIAL INPUT and SEQUENTIAL UPDATE may be positioned to a particular record within the data set by a READ 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. 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. (Refer to "GENKEY Option" on page 97 in Chapter 7.)

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, add, or replace records in an INDEXED data set. The following convention applies to record replacement.

The record specified by a source key in a REWRITE statement is replaced by the new record. If the data set contains FB-format records, a record replaced with a REWRITE statement causes an implicit READ statement to be executed unless the previous data transmission statement was a READ statement that obtained the record to be replaced.

Essential Information

To access an existing INDEXED data set, you must identify it to the operating system in DLBL and EXTENT statements, which must correspond with those used when the data set was created. The data set organization code ISE should be given in the DLBL statement when extending, accessing, or updating an INDEXED data set. If the data set is to be accessed for input or update in the same job step that created it, two separate PL/I files must be used and two complete sets of job control statements must be supplied. One set must have a DLBL statement with the code ISC and be associated with the PL/I file with the OUTPUT attribute. The other set must have a DLBL statement with the code ISE and be associated with the PL/I file with the INPUT or UPDATE attribute.

The essential information that must be supplied in a program for a file that accesses an INDEXED data set is similar to that described for the creation of an INDEXED data set. Certain additional ENVIRONMENT options can be used to improve performance when accessing and updating records in an INDEXED data set. These are the INDEXAREA, NOWRITE, and ADDBUFF options.

Note that an error could occur if a job fails when using INDEXAREA after the data set has been updated but before the modified cylinder index is written back onto the data set. If an INDEXED data set has an incorrect cylinder index for this reason, the entire data set should be reorganized as described below.

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.

There are two ways to reorganize an INDEXED data set:

- Copy the data set into a temporary CONSECUTIVE data set, and then re-create it in the original area of auxiliary storage.
- Copy the data set sequentially into a new area of auxiliary storage; you can then release the original auxiliary storage.

EXAMPLES OF INDEXED DATA SETS

Figure 51 on page 149 illustrates the creation of a simple INDEXED data set. The data set contains a telephone directory, using the subscribers' names as keys to the telephone numbers.

```
// JOB FIG0917
// OPTION LINK
// EXEC PLIOPT,SIZE=64K
 TELNOS: PROC OPTIONS(MAIN);
         DCL DIREC FILE RECORD SEQUENTIAL
             KEYED OUTPUT
             ENV(INDEXED KEYLENGTH(20) F
                 RECSIZE(3) MEDIUM(SYS006,
                 3330) VERIFY INDEXMULTIPLE
                 EXTENTNUMBER(3)
                 OFLTRACKS(2)),
             CARDIN RECORD INPUT
             ENV(MEDIUM(SYSIPT) F
                 RECSIZE(80)),
             CARD CHAR(80),
             NAME CHAR(20) DEFINED CARD,
             NUMBER CHAR(3) DEF CARD
              POS(21);
         ON ENDFILE(CARDIN) GO TO FINISH;
         OPEN FILE(DIREC),FILE(SYSPRINT),
              FILE(CARDIN);
 NEXTIN: READ FILE(CARDIN) INTO (CARD);
         PUT SKIP LIST (CARD);
         WRITE FILE(DIREC) FROM(NUMBER)
             KEYFROM(NAME);
         GO TO NEXTIN;
 FINISH: CLOSE FILE(DIREC),FILE(SYSPRINT),
             FILE(CARDIN);
         END TELNOS;
/*
// LBLTYP NSD(04)
// EXEC LNKEDT
// ASSGN SYS006,3330,VOL=DOS22,SHR
// DLBL DIREC,'TELNO',,ISC
// EXTENT SYS006,DOS222,4,0,3610,1
// EXTENT SYS006,DOS222,4,1,3611,4
// EXTENT SYS006,DOS222,1,2,3648,38
// EXTENT SYS006,DOS222,2,3,3686,19
// EXEC ,SIZE=64K
ACTION,G.           162
BAKER,R.            152
BRAMLEY,O.H.        248
CHEESEMAN,L.        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
/*
/&
```

Notes:

1. The VERIFY option causes the system to check that each record has been written correctly onto the direct access volume.
2. The EXTENTNUMBER option specifies that the data set TELNO is to consist of three extents. The extents are required for the master index, cylinder index, prime data area, and any independent overflow area. For the purpose of the EXTENTNUMBER option, the master and cylinder indexes are counted as one. However, for the LBLTYP option, they are counted as two.
3. The data set TELNO occupies 1 track for its master index, 4 tracks for the cylinder index, 38 tracks for the prime data area, and 19 tracks for the independent overflow area. The OFLTRACKS option reserves 2 tracks of each cylinder in the prime data area for cylinder overflow records.
4. LBLTYP is not required when operating under VSE/Advanced Functions.

Figure 51. Creating an INDEXED Data Set

```
// JOB FIG0918
// OPTION LINK
// EXEC PLIOPT,SIZE=64K
 DIRUPDT: PROC OPTIONS(MAIN);
          DCL DIREC FILE RECORD KEYED UPDATE DIRECT
              ENV(MEDIUM(SYS006,3330) KEYLENGTH(20) F RECSIZE(3)
              EXTENTNUMBER(3) VERIFY INDEXED
              INDEXMULTIPLE),
              CARDIN FILE RECORD INPUT ENV(MEDIUM(SYSIPT)
                                            F RECSIZE(80)),
              1 CARD,
              2 NAME CHAR(20),
              2 NEW_NUMBER CHAR(3),
              2 BLANK1 CHAR(1),
              2 OLD_NUMBER CHAR(3),
              2 BLANK2 CHAR(5),
              2 CODE CHAR(1),2 BLANK3 CHAR(47),
              ONCODE BUILTIN;
          ON ENDFILE(CARDIN) GO TO FINISH;
          ON KEY(DIREC) BEGIN;
             IF ONCODE = 51 THEN PUT FILE (SYSPRINT) EDIT
                        ('NAME NOT FOUND IN DATA SET')(X(5),A);
             IF ONCODE = 52 THEN PUT FILE (SYSPRINT) EDIT
                        ('DUPLICATE NAME') (X(5),A);
             END;
          OPEN FILE(DIREC), FILE (CARDIN),FILE(SYSPRINT);

 NEXT:    READ FILE(CARDIN) INTO (CARD);
          PUT FILE(SYSPRINT) SKIP EDIT(NAME,CODE)(A,X(2));
          IF CODE = 'A' THEN WRITE FILE (DIREC) FROM (NEW_NUMBER)
                                  KEYFROM(NAME);
          ELSE IF  CODE = 'C' THEN REWRITE FILE (DIREC) FROM
                        (NEW_NUMBER) KEY(NAME);
               ELSE PUT FILE (SYSPRINT)EDIT('CODE INVALID')(X(5),A);
          GO TO NEXT;

 FINISH:  CLOSE FILE(DIREC),FILE(CARDIN),FILE(SYSPRINT);
          END DIRUPDT;
/*
// LBLTYP NSD(04)
// EXEC LNKEDT
// ASSGN SYS006,3330,VOL=DOS222,SHR
// DLBL DIREC,'TELNO',,ISE
// EXTENT SYS006,DOS222,4,0,3610,1
// EXTENT SYS006,DOS222,4,1,3611,4
// EXTENT SYS006,DOS222,1,2,3648,38
// EXTENT SYS006,DOS222,2,3,3686,19
// EXEC ,SIZE=64K
NEWMAN,M.W.         516 450     C
LAW                 391         A
GOODFELLOW,D.T.     889         A
MILES,R.                233     C
HARVEY,C.D.W.       209         A
BARTLETT,S.G.       183         A
READ,K.M.           001         C
PITT,W.H.               515     X
ROLF,D.F.               114     C
ELLIOTT,D.          291 875     C
HASTINGS,G.M.           391     C
BRAMLEY,O.H.        439 248     C
/*
/&
```

Note: Deletion of unwanted telephone numbers is achieved by using the code C and rewriting the numbers as blanks.

Figure 52. Updating an INDEXED Data Set

The program in Figure 52 updates this data set. The input data includes codes to indicate the operations required:

A	Add a new record
C	Change an existing record (a blank new number indicates a lapsed account)

The link-editing steps in these examples include LBLTYP statements to specify that room will be required for processing the labels for the extents of the INDEXED data set. Under VSE/Advanced Functions, the LBLTYP statement is not required. Also included are ASSGN statements to associate the symbolic device name SYS006 with the 3330 disk drive that is to be used, DLBL statements to associate the file DIREC with the data set TELNO and specify its type and status (ISC or ISE), and EXTENT statements that give the symbolic device name, the disk storage volume (DOS222), the type of extent, and the locations.

REGIONAL DATA SETS

This section describes REGIONAL data set organization, data transmission statements, and the ENVIRONMENT options that define REGIONAL data sets. It then describes how to create and access REGIONAL(1) and REGIONAL(3) 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.

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. 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.

Direct access of REGIONAL data sets is quicker than that of INDEXED data sets, but it has 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.

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.

There are two types of REGIONAL data sets applicable to the DOS PL/I Optimizing Compiler, as described below.

Figure 53 on page 153 lists the data transmission statements and options that can be used to create and access a REGIONAL data set.

The DOS D PL/I compiler handles REGIONAL data sets differently, and allows an OUTPUT file to be used to add records to an existing data set. PL/I D compilers that take advantage of this facility should be modified before recompilation with the optimizing compiler.

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(option-list);
```

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
               ENVIRONMENT(option-list);
```

The file attributes are described in the OS and DOS PL/I Language Reference Manual.

ENVIRONMENT OPTIONS FOR REGIONAL DATA SETS

The following options apply to REGIONAL data sets.

```
  BLKSIZE         MEDIUM
  BUFFERS         RECSIZE
  COBOL           REGIONAL
  EXTENTNUMBER    SCALARVARYING
  F|U             VERIFY
  KEYLENGTH
```

The REGIONAL option is described in this chapter; the remaining options apply to two or more data set organizations and are described in Chapter 7. Figure 26 on page 96 summarizes the ENVIRONMENT options.

REGIONAL Option

The REGIONAL option defines a file with REGIONAL organization.

Syntax

REGIONAL([1|3])

1 | 3
specifies REGIONAL(1) 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 with a relative record within the data set (that is, region numbers start with 0 at the beginning of the 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);	KEYTO(reference)
	READ FILE(file-reference) SET(pointer-reference);	KEYTO(reference)
	READ FILE(file-reference) IGNORE(expression);	
SEQUENTIAL INPUT UNBUFFERED	READ FILE(file-reference) INTO(reference);	EVENT(event-reference) and/or KEYTO(reference)
	READ FILE(file-reference) IGNORE(expression);	EVENT(event-reference)
SEQUENTIAL UPDATE BUFFERED	READ FILE(file-reference) INTO(reference);	KEYTO (reference)
	READ FILE(file-reference) SET(pointer-reference);	KEYTO(reference)
	READ FILE(file-reference) IGNORE(expression);	
	REWRITE FILE(file-reference);	FROM(reference)
SEQUENTIAL UPDATE UNBUFFERED	READ FILE(file-reference) INTO(reference);	EVENT(event-reference) and/or KEYTO(reference)
	READ FILE(file-reference) IGNORE(expression);	EVENT(event-reference)
	REWRITE FILE(file-reference) FROM(reference);	EVENT(event-reference)
DIRECT OUTPUT	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	EVENT(event-reference)
DIRECT INPUT	READ FILE(file-reference) INTO(reference) KEY(expression);	EVENT(event-reference)

Figure 53 (Part 1 of 2). REGIONAL Data Set Statements and Options

File Declaration[1]	Valid Statements, 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)

[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);

Figure 53 (Part 2 of 2). REGIONAL Data Set Statements and Options

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. REGIONAL(3) data sets can be accessed by a file declared with REGIONAL(1) organization.

REGIONAL(3)
specifies that the data set contains F- or U-format records with recorded keys. Each region in the data set corresponds with a track on a direct access device and can contain one or more records.

Direct access of a REGIONAL(3) data set employs the region number specified in a source key to locate the required region. Once the region has been located, a sequential search is made for space to add a record, or for a record that has a recorded key identical with that supplied in the source key. Only one region (or track) is searched.

REGIONAL(1) organization is most suited to applications in which there are no duplicate region numbers, and where most of the regions will be filled (reducing wasted space in the data set). REGIONAL(3) is 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 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-string 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. Unlike the keys for INDEXED data sets, recorded keys in a REGIONAL data set are never embedded within the record.

Source Keys: The character-string 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-string 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 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; if the source key is longer than KEYLENGTH+8, the characters in the source key between the comparison key and the region number are ignored.

When generating the key, 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 ENVIRONMENT attribute has the option KEYLENGTH(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 option were KEYLENGTH(22), the search still would begin at the same place, but since the comparison key 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 KEYLENGTH(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 KEYLENGTH(15), the value of NAME is taken to be the comparison specification.

Reduction of Key Conversions: In general, for each input/output statement that specifies a source key, the compiler includes a conversion routine to convert the key to fixed binary. The conversion routine is not required if the following special cases are observed:

- For REGIONAL(1): When the source key is a fixed binary element variable or constant with precision (p,0), where 12 <= p <= 23.
- For REGIONAL(3): When the source key is of the form:

  ```
  (character-string-expression||r)
  ```

  where r is a fixed binary element variable or constant with precision (p,0), where 12 <= p <= 23.

REGIONAL(1) Organization

In a REGIONAL(1) data set, because there are no recorded keys, the region number serves as the sole identification of a particular record. The character-string 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 automatically inserted in the data set when it is created, 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 53 on page 153 shows the statements and options permitted 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 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 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.

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 53 shows the statements and options permitted 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 character string referenced in the KEYTO option has fewer than 8 characters, the value returned (the region number) is padded on the left with blanks; if it has more than 8 characters, it 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 (described early in this chapter).

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:

- <u>Retrieval</u>: All records, whether dummy or actual, are retrieved. The program must be prepared to recognize dummy records.
- <u>Addition</u>: A WRITE statement substitutes a new record for the existing record (actual or dummy) in the region specified by the source key.
- <u>Deletion</u>: The REWRITE statement can be used to delete a record by replacing it with a dummy record.
- <u>Replacement</u>: The record specified by the source key in a REWRITE statement, whether dummy or actual, is replaced.

REGIONAL(3) Organization

In a REGIONAL(3) 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. Each region can contain one or more records.

Each region number identifies a track on the direct access device that contains the data set; the region number should not exceed 16777215.

The data set can contain F- or U-format records. When a data set is preformatted, all tracks in the data set 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.

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 and continues to the end of that track.

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 53 on page 153 shows the statements and options permitted 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. If there is an error in the sequence, the KEY condition will be raised. If a track becomes filled by records for which the same region number was specified, the region number is automatically incremented by one; an attempt to add a further record with the same region number will raise 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 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.

ACCESSING A REGIONAL(3) DATA SET: After 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 53 shows the statements and options permitted 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 fewer characters than are specified in the

KEYLENGTH option, the value returned (the recorded key) is extended on the right with blanks; if it has more characters than specified by KEYLENGTH, 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.

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.

- <u>Retrieval</u>: The KEY condition is raised if a record with the specified recorded key is not found.
- <u>Addition</u>: A WRITE statement inserts the new record after any records already present on the specified track if space is available.
- <u>Replacement</u>: If the record specified by the source key in a REWRITE statement does not exist; the KEY condition is raised.

Note: If a track contains records with duplicate recorded keys, the record obtained when a duplicate key is specified is undefined.

ESSENTIAL INFORMATION FOR CREATING AND ACCESSING REGIONAL DATA SETS

To create a REGIONAL data set, you must supply a DLBL and EXTENT statement. The DLBL statement should include the data set organization code DA. Unless you are operating with VSE/Advanced Functions, a LBLTYP statement must be supplied when the program is link-edited so that main storage space is reserved for processing the data set labels.

When creating a REGIONAL data set, you must specify:

- The device type and symbolic device name of the direct access device that will contain the new data set, in the MEDIUM option.
- The record format and record size.
- The type of organization, either REGIONAL(1) or REGIONAL(3).
- For REGIONAL(3), you must also state the length of the recorded key in the KEYLENGTH option.

You can open an existing REGIONAL data set for sequential or direct access, and for input or update in each case. Using sequential input with a REGIONAL(1) data set, you can read all the records in ascending region number sequence, and in sequential update you can read and rewrite each record in turn. Sequential access of a REGIONAL(3) data set will give you the records in the order in which they appear in the data set. Using direct input, you can read any record by supplying a key containing its region number and, for REGIONAL(3), its recorded key; in direct update, you can read and rewrite existing records or add new ones. A PL/I program that processes a REGIONAL(1) data set must be able to recognize any dummy records that it encounters.

To access a REGIONAL data set, you must identify it to the operating system in a DLBL and EXTENT statement.

EXAMPLES OF REGIONAL DATA SETS

Included in these examples are LBLTYP statements to reserve space for processing a label for a REGIONAL data set, ASSGN statements for the non-standard device assignments that are used, DLBL statements to associate the files with the appropriate REGIONAL data sets, and EXTENT statements to define the symbolic device names, disk storage volume, and the locations within the volumes of the REGIONAL data sets. LBLTYP is not required when operating with VSE/Advanced Functions.

REGIONAL(1) Data Sets

Figure 54 and Figure 55 on page 161 illustrate the creation and updating of a REGIONAL(1) data set.

```
// JOB FIG0920
// OPTION LINK
// EXEC PLIOPT,SIZE=64K
 CRR1:   PROC OPTIONS(MAIN);
         DCL NOS FILE RECORD OUTPUT DIRECT KEYED
             ENVIRONMENT(F RECSIZE(20) REGIONAL(1) MEDIUM(SYS006,3330)),
             CARDIN FILE RECORD INPUT ENV(F RECSIZE(80) MEDIUM(SYSIPT)),
             CARD CHAR(80),
             NAME CHAR(20) DEF CARD,
             NUMBER CHAR(3) DEF CARD POS(21);
         ON ENDFILE(CARDIN) GO TO FINISH;
         OPEN FILE (NOS),FILE (CARDIN),FILE(SYSPRINT);
 NEXT:   READ FILE(CARDIN) INTO (CARD);
         PUT SKIP LIST(CARD);
         WRITE FILE(NOS) FROM(NAME) KEYFROM(NUMBER);
         GO TO NEXT;

 FINISH: CLOSE FILE(NOS),FILE(CARDIN),FILE(SYSPRINT);
         END CRR1;
/*
// LBLTYP NDS(01)
// EXEC LNKEDT
// ASSGN SYS006,3330,VOL=DOS222,SHR
// DLBL NOS,'NOSA',7,DA
// EXTENT SYS006,DOS222,1,0,3534,19
// EXEC ,SIZE=64K
ACTION,G.           162
BAKER,R.            152
BRAMLEY,O.H.        248
CHEESEMAN,L.        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 54. Creating a REGIONAL(1) Data Set

Figure 54 uses the same data as Figure 52 on page 150, but interprets it in a different way; the data set is effectively a list of telephone numbers with the names of the subscribers to whom they are allocated. The telephone numbers correspond to the region numbers in the data set, the data in each occupied region being a subscriber's name. Note that there are no recorded keys in a REGIONAL(1) data set.

The data read by the program in Figure 55 is identical with that used in Figure 53 on page 153, and the codes are interpreted in the same way. This program updates the data set, and then lists its contents. Before each new or modified record is written, the program tests the existing record in the region to ensure that it is a dummy; this is necessary because a WRITE statement can overwrite an existing record in a REGIONAL(1) data set even if it is not a dummy. Similarly, during the sequential reading and printing of the contents of the data set, each record is tested and dummy records are not printed.

```
// JOB FIG0921
// OPTION LINK
// EXEC PLIOPT,SIZE=64K
 ACR1:    PROC OPTIONS(MAIN);
          DCL NOS1 FILE KEYED RECORD DIRECT UPDATE
                 ENV(F RECSIZE(20) MEDIUM(SYS006,3330) REGIONAL(1)),
              NOS2 FILE KEYED RECORD INPUT
                 ENV(F RECSIZE(20) MEDIUM(SYS006,3330) REGIONAL(1)),
              NOS FILE VARIABLE,
              CARDIN FILE RECORD INPUT ENV(MEDIUM(SYSIPT)) F RECSIZE(80)),
              1 CARD,
                2 NAME CHAR(20),
                2 NEW_NUMBER CHAR(3),
                2 BLANK1 CHAR(1),
                2 OLD_NUMBER CHAR(3),
                2 BLANK2 CHAR(5),
                2 CODE CHAR(1),
                2 BLANK3 CHAR(47),
              IOFIELD CHAR(20),ONCODE BUILTIN,
              BYTE1 CHAR(1) DEF IOFIELD;
          ON ENDFILE(CARDIN) GO TO PRINT;
          ON KEY (NOS1) BEGIN;
          PUT EDIT ('NUMBER INVALID, ONCODE=',ONCODE)(X(5),A,A);
          GO TO NEXT;
          END;
          NOS=NOS1;
          OPEN FILE(NOS) TITLE('NOSA'), FILE (CARDIN);
 NEXT:    READ FILE(CARDIN) INTO (CARD);
          PUT FILE(SYSPRINT) SKIP EDIT(NAME(NEW_NUMBER,OLD_NUMBER,CODE)
                                  (A,X(2));
          IF CODE = 'A' THEN GOTO RITE;
           ELSE IF CODE = 'C' THEN DO;
             UNSPEC(BYTE1)=(8) '1'B;
             WRITE FILE(NOS) FROM (IOFIELD) KEYFROM(OLD_NUMBER);
             IF NEW_NUMBER='   ' THEN GOTO NEXT;
             ELSE GOTO RITE;
             END;
           ELSE PUT FILE (SYSPRINT) EDIT ('CODE INVALID')(X(5),A);
          GOTO NEXT;
 RITE:    READ FILE(NOS) KEY(NEW_NUMBER) INTO (IOFIELD);
          IF UNSPEC(BYTE1)=(8) '1'B THEN WRITE FILE(NOS)
                       KEYFROM(NEW_NUMBER) FROM(NAME);
          ELSE PUT FILE(SYSPRINT) EDIT ('NUMBER ALREADY USED')(X(5),A);
          GOTO NEXT;
 PRINT:   CLOSE FILE(NOS),FILE(CARDIN);
          PUT FILE(SYSPRINT) PAGE;
          NOS=NOS2;
          OPEN FILE(NOS) TITLE('NOSA');
          ON ENDFILE(NOS) GOTO FINISH;
```

Figure 55 (Part 1 of 2). Updating a REGIONAL(1) Data Set

```
 NEXTIN:  READ FILE(NOS) INTO(IOFIELD) KEYTO(NEW_NUMBER);
          IF UNSPEC(BYTE1) = (8) '1'B THEN GOTO NEXTIN;
          ELSE PUT FILE(SYSPRINT) SKIP EDIT (NEW_NUMBER,IOFIELD)
            (A,X(2)); GOTO NEXTIN;
 FINISH:  CLOSE FILE(NOS),FILE(SYSPRINT); END ACR1;
/*
// LBLTYP NSD(01)
// EXEC LNKEDT
// ASSGN SYS006,3330,VOL=DOS222,SHR
// DLBL NOSA,,'NOSA',,DA
// EXTENT SYS006,DOS222,1,0,3534,19
// EXEC ,SIZE=64K
NEWMAN,M.W.         516 450    C
LAW                 391        A
GOODFELLOW,D.T.     889        A
MILES,R.                233    C
HARVEY,C.D.W.       209        A
BARTLETT,S.G.       183        A
READ,K.M.           001        C
PITT,W.H.               515    X
ROLF,D.F.               114    C
ELLIOTT,D.          291 875    C
HASTINGS,G.M.           391    C
BRAMLEY,O.H.        439 248    C
/*
/&
```

Note: Deletion of unwanted records is achieved, in this example, by inserting X'FF' in the first data byte of each such record.

Figure 55 (Part 2 of 2). Updating a REGIONAL(1) Data Set

In this example, a file variable (NOS) has been used to enable records in a REGIONAL data set that has been opened for updating to be retrieved sequentially after completion of the update. This technique overcomes the restriction that a file cannot be declared with the attributes UPDATE and INPUT in the same program. Two files, NOS1 and NOS2, are declared and are assigned in turn to the file variable for processing. The TITLE option in both OPEN statements specifies the same identifier (NOSA) for the data set, thereby making one DLBL and EXTENT statement suffice for both files.

REGIONAL(3) Data Sets

Figure 56 on page 163 through Figure 58 on page 165 illustrate the use of REGIONAL(3) data sets. The figures depict a library processing scheme in which loans of books are recorded and reminders are issued for overdue books. Two data sets, STOCK3 and LOANS3, are involved. STOCK3 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.) LOANS3 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 LOANS3; the reader and book numbers are concatenated to form the recorded keys.

In Figure 56, the data sets STOCK3 and LOANS3 are created. The file LOANS, which is used to create the data set LOANS3, is opened for direct output merely to format the data set; the file is closed without any records being written onto the data set. It is assumed that the number of books on loan will not exceed 100; therefore the space operand in the EXTENT statement that defines LOANS3 requests 3 tracks, sufficient for 100 blocks of

19-byte records (12 bytes for data and a 7-byte key). The data set STOCK3 is created sequentially; duplicate region numbers are acceptable since each region can contain more than one record.

```
// JOB FIG0922
// OPTION LINK
// EXEC PLIOPT,SIZE=64K
 CRR3:  PROC OPTIONS(MAIN);
        DCL STOCK FILE RECORD SEQUENTIAL KEYED OUTPUT
             ENV(REGIONAL(3) MEDIUM(SYS006,3330) U RECSIZE(110) KEYLENGTH(4)),
             LOANS FILE RECORD DIRECT OUTPUT KEYED
             ENV(REGIONAL(3) MEDIUM(SYS005,3330) F RECSIZE(12) KEYLENGTH(7)),
            1 CARD,
              2 NUMBER PIC'9999',
              2 AUTHOR CHAR(25) VAR,
              2 TITLE CHAR(50) VAR,
              2 QTY1 FIXED DEC(3),
            1 BOOK,
              2 (L1,L2) FIXED DEC(3),
              2 ATY2 FIXED DEC(3),
              2 DESCN CHAR(75) VARYING,
            REGION PIC'99999999';
         ON ENDFILE(SYSIN) GO TO FINISH;
         OPEN FILE(LOANS), FILE (STOCK),FILE(SYSPRINT),FILE(SYSIN);
 NEXT:   GET FILE(SYSIN) LIST(CARD);
         L1 = LENGTH(AUTHOR);
         L2 = LENGTH(TITLE);
         QTY2=QTY1;
         DESCN = AUTHOR || TITLE;
         REGION = (NUMBER-1000)/1000;
         WRITE FILE (STOCK) FROM (BOO) KEYFROM(NUMBER||REGION);
         PUT SKIP EDIT (CARD) (A(4),X(1),A(25),X(1),A(50));
         GO TO NEXT;
 FINISH: CLOSE FILE(STOCK),FILE(LOANS),FILE(SYSPRINT),FILE(SYSIN);
         END CRR3;
/*
// LBLTYP NSD(01)
// EXEC LNKEDT
// ASSGN SYS005,3330,VOL=DOS222,SHR
// ASSGN SYS006,3330,VOL=DOS222,SHR
// DLBL LOANS,'LOANS3',,DA
// EXTENT SYS005,DOS222,1,0,3534,3
// DLBL STOCK,'STOCK3',,DA
// EXTENT SYS006,DOS222,1,0,3553,19
// EXEC ,SIZE=64K
'1015' 'J.M.BARRIE' 'PETER PAN'  1
'1214' 'L.CARROLL' 'THE HUNTING OF THE SNARK'  1
'3079' 'G.FLAUBERT' 'MADAME BOVARY'  1
'3083' 'V.HUGO' 'LES MISERABLES'  2
'3085' 'J.K.JEROME' 'THREE MEN IN A BOAT'  2
'4292' 'W.LANGLAND' 'THE BOOK CONCERNING PIERS THE PLOWMAN'  1
'5999' 'W.SHAKESPEARE' 'MUCH ADO ABOUT NOTHING'  3
'6591' 'F.RABELAIS' 'THE HEROIC DEEDS OF GARGANTUA AND PANTAGRUEL'  1
'8326' 'H.D.THOREAU' 'WALDEN, OR LIFE IN THE WOODS'  1
'9765' 'N.MELLERSH' 'THE DISCOVERERS OF THE UNIVERSE'  3
/*
/&
```

Figure 56. Creating a REGIONAL(3) Data Set

Figure 57 on page 164 illustrates the updating of the data set LOANS3. 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 position of the reader number on the card allows the 8-character region number to be derived directly by overlay defining. The DATE built-in function is used to obtain the current data. This data is written in both the issue-data and reminder-data portions of a new record or an updated record.

The region number for the data set LOANS3 is obtained by testing the reader number.

```
// JOB FIG0923
// OPTION LINK
// EXEC PLIOPT,SIZE=64K
 DUR3:  PROC OPTIONS(MAIN);
        DCL 1 RECORD,
              2 (ISSUE,REMINDER) CHAR(6),
            SYSIN FILE RECORD INPUT SEQUENTIAL ENV(F RECSIZE(80) MEDIUM(SYSIPT)),
            LOANS FILE RECORD UPDATE DIRECT KEYED
               ENV(REGIONAL(3) F RECSIZE(12) KEYLENGTH(7) MEDIUM(SYS006,3330)),
            CARD CHAR(80),
            BOOK CHAR(4) DEFINED CARD,
            READER CHAR(3) DEFINED CARD POSITION(10),
            CODE CHAR(1) DEFINED CARD POSITION(20),
            RCODE CHAR(1) DEFINED RECORD,
            REGION CHAR(8),
            DATE BUILTIN;
          ON ENDFILE(SYSIN) GO TO FINISH;
          OPEN FILE(SYSIN),FILE(LOANS),FILE(SYSPRINT);
          ISSUE,REMINDER = DATE();
 NEXT:    READ FILE(SYSIN) INTO (CARD);
          PUT FILE(SYSPRINT) SKIP EDIT(CARD)(A(20));
   /* '****' IS A DUMMY TO SEPARATE ENTRIES FOR DIFFERENT DAYS */
          IF BOOK = '****' THEN GO TO NEXT'
          IF READER < '034' THEN REGION = (8) '0';
             ELSE IF READER < '067' THEN REGION ='00000001';
             ELSE REGION = '00000002';
          IF CODE = 'I' THEN DO;
             WRITE FILE(LOANS) FROM(RECORD) KEYFROM(READER||BOOK||REGION);
             GOTO NEXT;
             END;
          IF CODE = 'R' THEN DO;
             UNSPEC(RCODE) = (8) '1'B;
             REWRITE FILE (LOANS) FROM (RECORD) KEY (READER || BOOK || REGION);
             END;
          ELSE IF CODE = 'A' THEN REWRITE FILE(LOANS) FROM (RECORD)
                                  KEY (READER || BOOK || REGION);
          ELSE PUT FILE (SYSPRINT) EDIT ('CODE INVALID')(X(5),A);
          GOTO NEXT;
 FINISH:  CLOSE FILE(SYSIN),FILE(LOANS),FILE(SYSPRINT);
          END DUR3;
/*
// LBLTYP NSD(01)
// EXEC LNKEDT
// ASSGN SYS006,3330,VOL=DOS222,SHR
// DLBL LOANS,'LOANS3',,DA
// EXTENT SYS006,DOS222,1,0,3534,3
// EXEC ,SIZE=64K
**** MONDAYS DATA
5999    003       I
3083    091       I
1214    095       I
**** TUESDAYS DATA
5999    003       I
3083    091       R
3517    095       X
/*
/&
```

Figure 57. REGIONAL(3) Data Sets: Direct Update

```
// JOB FIG0924
// OPTION LINK
// EXEC PLIOPT,SIZE=64K
 SUR3:   PROCEDURE OPTIONS(MAIN);
         DCL LOANS FILE RECORD SEQUENTIAL UPDATE KEYED
                   ENV(REGIONAL(3) F RECSIZE(12) KEYLENGTH(7) MEDIUM(SYS005,3330)),
             STOCK FILE RECORD DIRECT INPUT KEYED
                   ENV(REGIONAL(3) U RECSIZE(110) KEYLENGTH(4) MEDIUM(SYS006,3330)),
             (TODAY,LASMTH) CHAR(6),
             YEAR PIC '99' DEF LASMTH,
             MONTH PIC '99' DEF LASMTH POS(3),
             1 RECORD,
               2 (ISSUE,REMINDER) CHAR(6),
             DCODE CHAR(1) DEF ISSUE,
             LOANKEY CHAR (7),
             READER CHAR (3) DEFINED LOANKEY,
             BKNO CHAR (4) DEF LOANKEY POS(4),
             INTER FIXED DEC(5),
             REGION CHAR(8),
             1 BOOK,
               2 (L1,L2) FIXED DEC(3),
               2 QTY FIXED DEC(3),
               2 DESCN CHAR(75) VAR,
             AUTHOR CHAR(25) VAR,
             TITLE CHAR(50) VAR,DATE BUILTIN;
         TODAY,LASMTH = DATE();
         IF MONTH = '01' THEN DO;
            MONTH = '12';
            YEAR = YEAR-1;
            END;
         ELSE MONTH = MONTH-1;
         OPEN FILE(LOANS),FILE(STOCK),FILE(SYSPRINT);
         ON ENDFILE(LOADS) GOTO FINISH;
 NEXT:   READ FILE (LOANS) INTO (RECORD) KEYTO(LOANKEY);
         IF UNSPEC(DCODE)='11111111' THEN GOTO NEXT;
         IF REMINDER < LASMTH THEN DO;
            REMINDER = TODAY;
            REWRITE FILE(LOANS) FROM(RECORD);
            INTER = (BKNO-1000)/1000;
            REGION = INTER;
            READ FILE(STOCK) INTO (BOOK) KEY(BKNO||REGION);
            AUTHOR = SUBSTR(DESCN,1,L1);
            TITLE = SUBSTR(DESCN,L1+1,L2);
            PUT FILE(SYSPRINT) SKIP(4) EDIT(READER,AUTHOR,TITLE) (A,SKIP(2));
            END;
         GO TO NEXT;
 FINISH: CLOSE FILE(LOANS),FILE(STOCK),FILE(SYSPRINT);
         END SUR3;
/*
// LBLTYP NSD(01)
// EXEC LNKEDT
// ASSGN SYS005,3330,VOL=DOS222,SHR
// ASSGN SYS006,3330,VOL=DOS222,SHR
// DLBL LOANS,'LOANS3',,DA
// EXTENT SYS005,DOS222,1,0,3534,3
// DLBL STOCK,'STOCK3',,DA
// EXTENT SYS006,DOS222,1,0,3553,19
// EXEC ,SIZE=64K
/&
```

Figure 58. REGIONAL(3) Data Sets: Sequential Update and Direct Input

In order to make the REGIONAL(3) examples testable without introducing a further job, it is assumed that several days' input will be presented at once. In the example, Monday's and Tuesday's loans and returns are presented together, allowing the R (return) function to be tested.

The program in Figure 58 uses a sequential update file (LOANS) to process the records in the data set LOANS3, and a direct input file (STOCK) to obtain the book description from the data set STOCK3 for use in the reminder note. Each record from LOANS3 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.

To conserve space in the data set STOCK3, U-format records are used. In each record, the author's name and the title of the book are concatenated in a single character string, and the lengths of the two parts of the string are written as part of the record.

CHAPTER 10. USING VSAM DATA SETS FROM PL/I

This chapter describes VSAM (the Virtual Storage Access Method) organization for record-oriented transmission, the VSAM ENVIRONMENT options, compatibility with other PL/I data set organizations, and the statements used to load and access the three types of VSAM data sets—entry-sequenced, key-sequenced, and relative record. The chapter concludes with a series of examples showing the PL/I statements, Access Method Services commands, and job control statements necessary to create and access VSAM data sets. These are supported by short descriptions.

Appendix B introduces VSAM and Access Method Services. It describes the commands for defining and deleting data sets, and for building alternate indexes.

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 job control 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 <u>control intervals</u>, which are in turn arranged in <u>control areas</u>. 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 <u>prime index</u> 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. Key-sequenced and entry-sequenced data sets can both have one or more <u>alternate</u> indexes created for them. 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. 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. 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 must, however, 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.

Figure 59 on page 170 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 these options in Chapter 12 of the OS and DOS PL/I Language Reference Manual. See also "Embedded Keys" on page 137 in Chapter 9 of this book.

Relative Byte Addresses (RBA)

Relative byte addresses allow you to use keyed access on an ESDS associated with a KEYED SEQUENTIAL file. The RBAs, or keys, are character strings of length 4, and their values are defined by VSAM. 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 the use of the relative byte address as a key to a KSDS, but this is not supported by PL/I.

Relative Record Numbers

Records in an RRDS are identified by a relative-record number that starts at 1 and is incremented by 1 for each succeeding record. 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 9, there are three types of VSAM data sets and four 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 liable to performance degradation through misuse of function.

The comparison of the data set types given in Figure 35 on page 118 in Chapter 9 is helpful; however, many factors in the choice of data set type for a large installation are beyond the scope of this book.

Figure 59 on page 170 shows 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.

The diagrams show how the information contained in the family tree below could be held in VSAM data sets of different types.

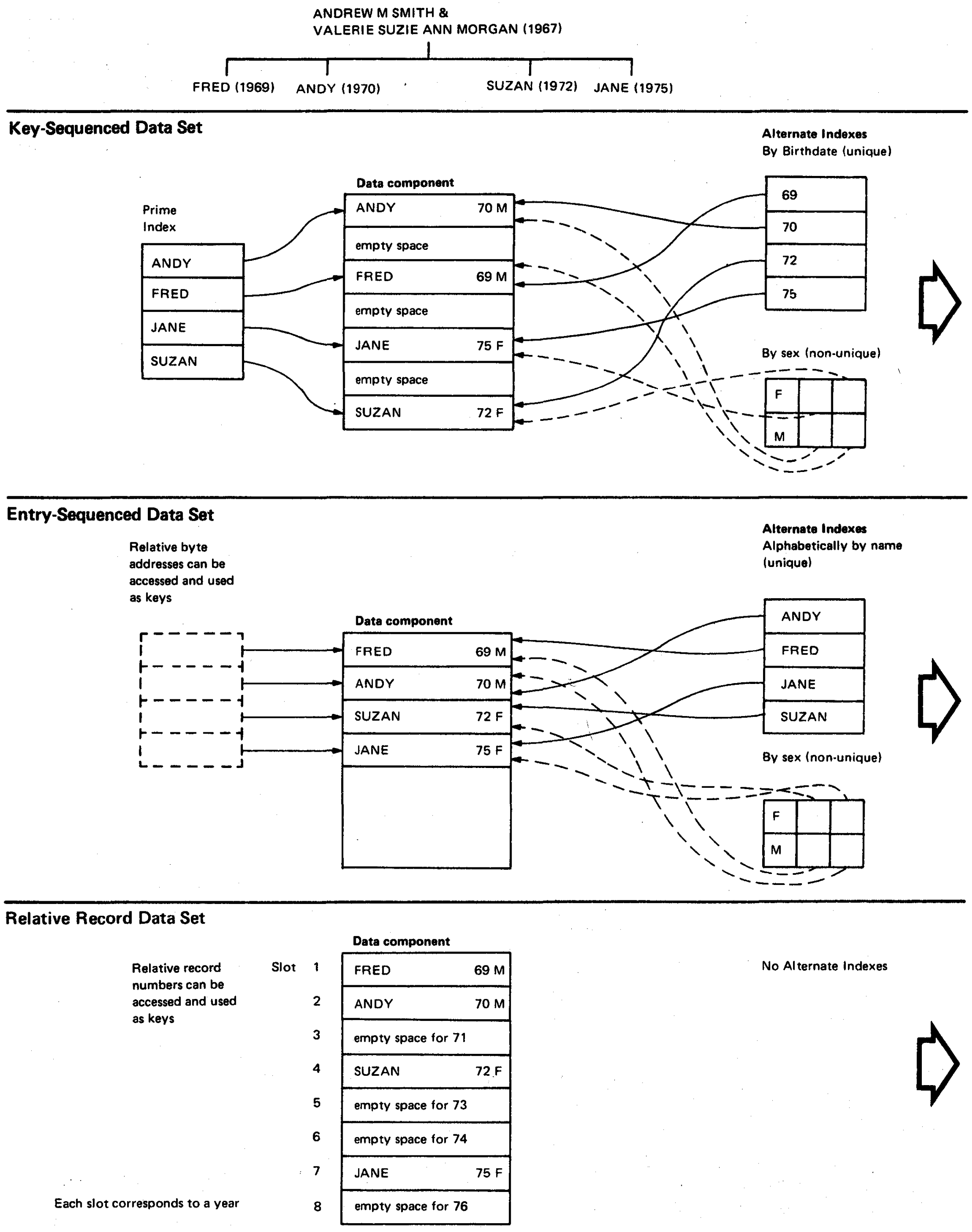


Figure 59 (Part 1 of 2). Types and Advantages of VSAM Data Sets

	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 59 (Part 2 of 2). Types and Advantages of VSAM Data Sets

1. Determine the type of data and how it will be accessed.
   - Primarily sequentially—favors ESDS
   - Primarily by key—favors KSDS
   - 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 limits 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 60 on page 173 and Figure 61 on page 179 may be helpful in this determination.

Figure 62 on page 180 through Figure 64 on page 186 show the statements permitted for entry-sequenced data sets, INDEXED data sets, and relative-record data sets, respectively.

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(option-list);
```

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(option-list);
```

Figure 25 on page 95 in Chapter 7 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 60 on page 173 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	REUSE
BKWD	SKIP
BUFND	COBOL
BUFNI	GENKEY
BUFSP	SCALARVARYING
PASSWORD	

COBOL, GENKEY, and SCALARVARYING have the same effect as for non-VSAM data sets.

	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	Path(N)	Alternate index path with nonunique keys (See "Alternate Index Paths" in Appendix B for details.)
KSDS	Key-sequenced data set		
RRDS	Relative-record data set		
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.

Figure 60. VSAM Data Sets and Permitted File Attributes

The options checked for a VSAM data set are RECSIZE, and, for a key-sequenced data set, KEYLENGTH and KEYLOC.

Figure 26 on page 96 in Chapter 7 shows which options are ignored for VSAM, as well as the required and default options.

VSAM Option

Specify the VSAM option for VSAM data sets.

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. After that 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 access method 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 B. Specify the highest level of password needed for the type of access that your program will perform.

GENKEY Option

For the description of the GENKEY option, see "GENKEY Option" on page 97 in Chapter 7.

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 key 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, BUFND, BUFNI, and BUFSP are options you can specify to optimize VSAM's performance.

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 be 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.

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 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 whenever 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.

COMPATIBILITY WITH OTHER DATA SET ORGANIZATIONS

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 DOS/VS Access Method Services User's Guide.

- All VSAM key-sequenced data sets have embedded keys, even if they have been converted from ISAM data sets with nonembedded keys.

- The use of programs written for non-VSAM data sets with VSAM data sets without alteration of the programs. This is described in the next section under "The VSAM Compatibility Interface."

- The alteration of existing programs to allow them to use VSAM data sets. A discussion of this is given at the end of this section.

THE VSAM COMPATIBILITY INTERFACE

The VSAM 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 needed in the following circumstances:

- If your program uses nonembedded keys.
- If your program relies on the raising of the RECORD condition when an incorrect-length record is encountered.
- 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.

If any of these conditions apply, the program cannot satisfactorily be written for VSAM without the compatibility interface.

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. Programs with REGIONAL(1) data sets require only minor revision.

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 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 Files

Programs using REGIONAL(1) data sets cannot be used unaltered to access 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 the VSAM ENVIRONMENT option.

Any checking for deleted records should be removed because VSAM deleted records are not accessible to you.

Programs with REGIONAL(3) files will need restructuring before they can be used with VSAM data sets.

ASSOCIATING SEVERAL VSAM FILES WITH ONE DATA SET

The TITLE option of the OPEN statement can be used to associate two or more PL/I files with the same VSAM data set in the manner described in Chapter 7 for non-VSAM data sets. PL/I creates one set of control blocks (an Access Method Control Block (ACB) and a Request Parameter List (RPL)) for each file, and does not provide the facility to associate 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.

- Other files may retrieve down-level records. This can be avoided by opening all files 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. The only way to avoid this error is to ensure that the two files are not accessing the same control interval in the data set. You can design your program to retry the retrieval after completion of the other file's data transmission.
- 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-partition or cross-system sharing of data sets. These types of sharing are discussed in Appendix B and further described in the DOS/VS Access Methods Services User's Guide.

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 VSAM Data Sets" on page 304 in Appendix B).

You can then write the required DLBL statement to access the data set in the form:

```
  // DLBL filename,'data set name',,VSAM
```

For example, if your file is is called PL1FILE, your data set called VSAMDS, and your data set is on volume DOS222, you would enter:

```
  // DLBL PL1FILE,'VSAMDS',,VSAM
  // EXTENT SYS006,DOS222
```

ASSOCIATING AN ALTERNATE INDEX PATH WITH A FILE

When using an alternate index, you simply specify the name of the path as the data set name (file identifier) of the DLBL statement associating the base data set/alternate index pair with your PL/I file. Before using an alternate index, you should be aware of the restrictions on processing; these are summarized in Figure 61 on page 179. The method used for defining a path and building an alternate index is given in Appendix B.

Assuming that a PL/I file was called PLIFILE and the alternate index path was called PERSALPH, and that it was on volume DOS222, the DLBL and EXTENT statements could take the form:

```
// DLBL PLIFILE,'PERSALPH',,VSAM
// EXTENT SYS006,DOS222
```

```
******************************************************
```

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 61. Processing Allowed on Alternate Indexes

ENTRY-SEQUENCED DATA SETS

The statements and options allowed for files associated with an ESDS are shown in Figure 62 on page 180.

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)
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]
(See notes in Part 2.)		

Figure 62 (Part 1 of 2). VSAM Entry-Sequenced Data Set Statements and Options

[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.

Figure 62 (Part 2 of 2). VSAM Entry-Sequenced Data Set Statements and Options

KEY-SEQUENCED AND INDEXED ENTRY-SEQUENCED DATA SETS

The statements and options permitted for INDEXED VSAM data sets are shown in Figure 63 on page 182. 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 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, regardless of the position of any previous access.

If the data set is being accessed via a unique key 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 key 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.

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-reference)[5]	KEY(expression)
SEQUENTIAL UPDATE UNBUFFERED	READ FILE(file-reference) INTO(reference);	EVENT(event-ref.) and/or either KEY(expression) or KEYTO(reference)
	READ FILE(file-reference);[2]	EVENT(event-reference) and/or IGNORE(expression)
(See notes in Part 3.)		

Figure 63 (Part 1 of 3). VSAM INDEXED Data Set Statements and Options

File Declaration[1]	Valid Statements, with Options that Must Appear	Other Options that Can also Be Used
	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[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);	
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 63 (Part 2 of 3). VSAM INDEXED Data Set Statements and Options

[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.

[2]The statement: READ FILE(file-reference); is equivalent to the statement: READ FILE(file-reference) IGNORE(1);

[3]A SEQUENTIAL OUTPUT 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 key alternate index.

[5]DELETE statements are not allowed for a file associated with an ESDS accessed via an alternate index.

Figure 63 (Part 3 of 3). VSAM INDEXED Data Set Statements and Options

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 key index.

If a DIRECT OUTPUT file is used to add records to the data set, the KEY condition is raised if an attempt is made to insert a record with the same key as a record that already exists.

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.

RELATIVE-RECORD DATA SETS

The statements and options permitted for VSAM relative record data sets (RRDS) are shown in Figure 64 on page 186.

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 168).

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 over.

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, regardless of the position of any previous access. For WRITE with the KEYFROM option, the KEY condition is raised if an attempt is made to insert a record with the same relative-record number as a record that already exists on the data set. If 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.

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.

EXAMPLES WITH ENTRY-SEQUENCED DATA SETS

The examples in Figure 65 on page 188 through Figure 69 on page 191 for ESDS are based on the family tree shown in Figure 59 on page 170.

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);	KEYFROM(expression) or KEYTO(reference)
	LOCATE based-variable FILE(file-reference);	SET(pointer-reference) and/or KEYFROM(expression)
SEQUENTIAL OUTPUT UNBUFFERED	WRITE FILE(file-reference) FROM(reference);	EVENT(event-reference) and/or either KEYFROM(expression) or KEYTO(reference)
SEQUENTIAL INPUT BUFFERED	READ FILE(file-reference) INTO(reference);	KEY(expression) or KEYTO(reference)
	READ FILE(file-reference) SET(pointer-reference);	KEY(expression) or KEYTO(reference)
	READ FILE(file-reference);[2]	IGNORE(expression)
SEQUENTIAL INPUT UNBUFFERED	READ FILE(file-reference) INTO(reference);	EVENT(event-ref.) and/or either KEY(expression) or KEYTO(reference)
	READ FILE(file-reference);[2]	EVENT(event-reference) and/or IGNORE(expression)
SEQUENTIAL UPDATE BUFFERED	READ FILE(file-reference) INTO(reference);	KEY(expression) or KEYTO(reference)
	READ FILE(file-reference) SET(pointer-reference);	KEY(expression) or KEYTO(reference)
	READ FILE(file-reference);[2]	IGNORE(expression)
	WRITE FILE(file-reference) FROM(reference);	KEYFROM(expression) or KEYTO(reference)
	REWRITE FILE(file-reference);	FROM(reference) and/or KEY(expression)
SEQUENTIAL UPDATE UNBUFFERED	READ FILE(file-reference) INTO(reference);	EVENT(event-ref.) 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-ref.) and/or either KEYFROM(expression) or KEYTO(reference)

Figure 64 (Part 1 of 2). VSAM Relative-Record Data Set Statements and Options

File Declaration[1]	Valid Statements, with Options that Must Appear	Other Options that Can also Be Used
SEQUENTIAL UPDATE UNBUFFERED, cont.	REWRITE FILE(file-reference) FROM(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);	
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)

[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.

[2]The statement: READ FILE(file-reference); is equivalent to the statement: READ FILE(file-reference) IGNORE(1);

Figure 64 (Part 2 of 2). VSAM Relative-Record Data Set Statements and Options

Defining and Loading an Entry-Sequenced Data Set

In Figure 65, the data set is defined with the DEFINE CLUSTER command and given the name SMITHFAM.BASE. The NONINDEXED keyword causes an ESDS to be defined.

```
// JOB DOS10#7 A.J.COS,N081,H205434
// OPTION CATAL
 PHASE PGMA,*
// EXEC IDCAMS,SIZE=30K
     DEFINE CLUSTER -
      (NAME(PL1VSAM.AJC1.BASE) -
      VOLUMES(DOS111) -
      NONINDEXED -
      RECORDSIZE(80 80) -
      TRACKS(2 2))
/*
// EXEC PLIOPT, SIZE=64K
   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);
        WRITE FILE(FAMFILE) FROM (STRING);
      END;
FINITO:
      PUT SKIP EDIT(I-1,' RECORDS PROCESSED')(A);
    END;
/*
// EXEC LNKEDT
// DLBL FAMFILE,'PL1VSAM.AJC1.BASE',,VSAM
// EXTENT SYS006,DOS111
// ASSGN SYS006,3330,VOL=DOS111,SHR
// EXEC PGMA,SIZE=60K
FRED                         69          M
ANDY                         70          M
SUZAN                        72          F
/*
/&
```

Figure 65. 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 DLBL statement for the file contains the data set name (file identifier) 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 OPTION CATAL and a PHASE statement 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 Figure 65.

Updating an Entry-Sequenced Data Set

Figure 66 shows the addition of a new record on the end of an ESDS. This is done by reexecuting the program shown in Figure 65. A SEQUENTIAL OUTPUT file is used and the data set associated with it by use of the data set name (file identifier) specifying the name PL1VSAM.AJC1.BASE specified in the DEFINE command shown in Figure 65.

Existing records can be rewritten in an ESDS provided that the length of the record is not changed. A SEQUENTIAL or 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 entry-sequenced data sets.

```
// JOB DOS10#8 A.J.COS,N081,H105434
// DLBL FAMFILE,'PL1VSAM.AJC1.BASE',0,VSAM
// EXTENT SYS006,DOS111
// ASSGN SYS006,3330,VOL=DOS111,SHR
// EXEC PGMA,SIZE=60K
JANE                        75         F
/*
/&
```

Figure 66. Updating an ESDS

Creating a Unique Key Alternate Index Path for an ESDS

Figure 67 shows the creation of a unique key alternate index path for the ESDS defined and loaded in Figure 65. Using this path, the data set is indexed by the name of the child in the first 15 bytes of the record.

```
// JOB DOS10#9 A.J.COX,N081,H205434
// EXEC IDCAMS,SIZE=30K
         DEFINE ALTERNATEINDEX -
           (NAME(PL1VSAM.AJC1.ALPHIND) -
            VOLUMES(DOS111) -
            TRACKS(4 1) -
            KEYS(15 0) -
            RECORDSIZE(20 40) -
            UNIQUEKEY -
            RELATE(PL1VSAM.AJC1.BASE))
/*
// DLBL DD1,'PL1VSAM.AJC1.BASE',0,VSAM
// EXTENT SYS006,DOS111
// DLBL DD2,'PL1VSAM.AJC1.ALPHIND',0,VSAM
// EXTENT SYS006,DOS111
// ASSGN SYS006,3330,VOL=DOS111,SHR
// EXEC IDCAMS,SIZE=60K
         BLDINDEX INFILE(DD1) OUTFILE(DD2)

         DEFINE PATH -
           (NAME(PL1VSAM.AJC1.ALPHPATH) -
            PATHENTRY(PL1VSAM.AJC1.ALPHIND))
/*
/&
```

Figure 67. Creating a Unique Key Alternate Index Path for an ESDS

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 DLBL and EXTENT statements.

DLBL and EXTENT 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 more complete description of defining an alternate index is given in Appendix B on page 297.

Creating a Nonunique Key Alternate Index Path for an ESDS

Figure 68 shows the creation of a nonunique key alternate index path for an ESDS. The alternate index enables the data to be selected by the gender 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.

```
// JOB DOS10#10 A.J.COX,NO81 H205434
// EXEC IDCAMS,SIZE=30K

     DEFINE ALTERNATEINDEX -
       (NAME(PL1VSAM.AJC1.SEXIND) -
        VOLUMES(DOS111) -
        TRACKS(4 1) -
        KEYS(1 37) -
        NONUNIQUEKEY -
        RELATE(PL1VSAM.AJC1.BASE)) -
        RECORDSIZE(20 400))
/*
// DLBL DD1,'PL1VSAM.AJC1.BASE',,VSAM
// EXTENT SYS006,DOS111
// ASSGN SYS006,3330,VOL=DOS111,SHR
// EXEC IDCAMS,SIZE=30K
    BLDINDEX INFILE(DD1) OUTFILE(DD2)

    DEFINE PATH -
      (NAME(PL1VSAM.AJC1.SEXPATH) -
       PATHENTRY(PL1VSAM.AJC1.SEXIND))
/*
/&
```

Figure 68. Creating a Nonunique Key Alternate Index Path on an ESDS

The three commands and the DLBL and EXTENT statements are as described in Figure 67 on page 189. 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 a large enough RECORDSIZE is specified. 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 69 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.
GENDFLE	is the alternate index path that corresponds to the gender of the children.

There are DLBL and EXTENT statements for all of the files. They connect BASEFLE and BACKFLE to the base data set by specifying the name of the base data set as the data set name (file identifier), and connect ALPHFLE and GENDFLE by specifying the names of the paths given in Figure 67 on page 189 and Figure 68 on page 190.

```
// JOB DOS10#11 A.J.COX,N081,H205434
// OPTION LINK
// EXEC PLIOPT, SIZE=64K
  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*/
          GENDFIL 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 GENDER 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);
        PUT SKIP(2);
    DO WHILE('1'B);
       READ FILE(BASEFLE) INTO (STRING);
       PUT SKIP EDIT(STRING)(A);
    END;
  YPRINT:
    CLOSE FILE(BASEFLE);
     /*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);
    PUT SKIP(2);

    DO WHILE('1'B);
      READ FILE(BACKFLE) INTO (STRING);
      PUT SKIP EDIT(STRING)(A);
    END;

  AGEQUERY:  CLOSE FILE(BACKFLE);
```

Figure 69 (Part 1 of 2). Alternate Index Paths and Backward Reading with an ESDS

```
    /*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);

   /*Use the alternate index to print out all the girls in the
     family*/
        ON ENDFILE(GENDFIL) GOTO FINITO;

        PUT SKIP(2) EDIT('ALL THE GIRLS')(A);
        PUT SKIP(2);

        READ FILE(GENDFIL) INTO (STRING) KEY('F');
        PUT SKIP EDIT(STRING)(A);
        DO WHILE(SAMEKEY(GENDFIL));
           READ FILE(GENDFIL) INTO (STRING);
           PUT SKIP EDIT(STRING)(A);
        END;
FINITO:
        END;
/*
// EXEC LNKEDT
// DLBL BASEFLE,'PL1VSAM.AJC1.BASE',0,VSAM
// EXTENT SYS006,DOS111
// DLBL BACKFLE,'PL1VSAM.AJC1.BASE',0,VSAM
// EXTENT SYS006,DOS111
// DLBL ALPHFLE,'PL1VSAM.AJC1.ALPHPATH',0,VSAM
// EXTENT SYS006,DOS111
// DLBL GENDFIL,'PL1VSAM.AJC1.GENDPATH',0,VSAM
// EXTENT SYS006,DOS111
// ASSGN SYS006,3330,VOL=DOS111,SHR
// EXEC ,SIZE=60K
ANDY
/*
// EXEC IDCAMS,SIZE=30K
     DELETE -
          PL1VSAM.AJC1.BASE
/*
/&
```

Figure 69 (Part 2 of 2). Alternate Index Paths and Backward Reading with an ESDS

The program uses SEQUENTIAL files to access the data and print it first in the normal order then in the reverse order. At the label AGEQUERY a DIRECT file is used to read the data associated with an alternate index key in the unique alternate index.

Finally, at the label SPRINT a KEYED SEQUENTIAL file is used to print a list of the females in the family using the nonunique key alternate index path. The SAMEKEY built-in function is used to read all the records with the same key. The females will be accessed in the order in which their names 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 Appendix B on page 297.

EXAMPLES WITH KEY-SEQUENCED DATA SETS

The examples in Figure 70 on page 194 through Figure 73 on page 197 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 70 the data set is defined and loaded. In Figure 71 it is updated by means of a prime index. In Figure 73 a unique key alternate index path is created using the numbers as the alternate key. In Figure 73, 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. They can be compared with the INDEXED data set examples in Chapter 9.

Defining and Loading a Key-Sequenced Data Set

Figure 70 on page 194 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 option. The position of the keys within the record is defined in the KEYS option.

Within the PL/I program a KEYED SEQUENTIAL OUTPUT file is used with a WRITE...FROM...KEYFROM statement. The data is presented in ascending key order. A KSDS must be loaded in this manner.

The file is associated with the data set by a DLBL statement that uses the name given in the DEFINE command as the data set name (file identifier), using the prime index.

Updating a Key-Sequenced Data Set

Figure 71 on page 195 shows one method by which a KSDS can be updated.

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

```
// JOB DOS10#12 A.J.COS,N081,H205434
// OPTION LINK
// EXEC IDCAMS,SIZE=30K

   DEFINE CLUSTER -
     (NAME(PL1VSAM.AJC2.BASE) -
     VOLUMES(DOS111) -
     INDEXED -
     TRACKS(3 1) -
     KEYS(20 0) -
     RECORDSIZE(23 80))
/*
// EXEC PLIOPT, SIZE=64K
 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;

/*
// EXEC LNKEDT
// DLBL DIREC,'PL1VSAM.AJC2.BASE',0,VSAM
// EXTENT SYS006,DOS111
// ASSGN SYS006,3330,VOL=DOS111,SHR
// EXEC ,SIZE=64K
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 70. Defining and Loading a Key-Sequenced Data Set (KSDS)

```
// JOB DOS10#13 A.J.COX,N081,H205434
// OPTION LINK
// EXEC PLIOPT, SIZE=64K
 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)(A(20),A(3),X(56),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;
/*
// EXEC LNKEDT
// DLBL DIREC,'PL1VSAM.AJC2.BASE',0,VSAM
// EXTENT SYS006,DOS111
// ASSGN SYS006,3330,VOL=DOS111,SHR
// EXEC ,SIZE=64K
NEWMAN,M.W.         516                                                      C
GOODFELLOW,D.T.     889                                                      A
MILES,R.                                                                     D
HARVEY,C.D.W.       209                                                      A
BARTLETT,S.G.       183                                                      A
CORY,G.                                                                      D
READ,K.M.           001                                                      A
PITT,W.H.
ROLF,D.F.                                                                    D
ELLIOTT,D.          291                                                      C
HASTINGS,G.M.                                                                D
BRAMLEY,O.H.        439                                                      C
/*
/&
```

Figure 71. Updating a KSDS

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 DLBL statement that uses the data set name (file identifier) PL1VSAM.AJC2.BASE, defined in the DEFINE CLUSTER command in Figure 70 on page 194.

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 freespace 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 72.

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 freespace 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.

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

Figure 72. VSAM Methods of Insertion into a Key-Sequenced Data Set

Creating a Unique Key Alternate Index Path for a KSDS

Figure 73 on page 197 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 numeric order to show what 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 DLBL and EXTENT statements.

DLBL and EXTENT 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 Appendix B on page 297.

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 68 on page 190 for an example of the creation of an alternate index with nonunique keys.

```
// JOB DOS10#14 A.J.COS,N081,H205434
// EXEC IDCAMS,SIZE=30K
      DEFINE ALTERNATEINDEX -
        (NAME(PL1VSAM.AJC2.NUMIND) -
         VOLUMES(DOS111) -
         TRACKS(4 4) -
         KEYS(3 20) -
         RELATE(PL1VSAM.AJC2.BASE) -
         UNIQUEKEY -
         RECORDSIZE(24 48)) -
       CATALOG(DOS111.VSAMCAT)
/*
// DLBL DD1,'PL1VSAM.AJC2.BASE',0,VSAM
// EXTENT SYS006,DOS111
// DLBL DD2,'PL1VSAM.AJC2.NUMIND',0,VSAM
// EXTENT SYS006,DOS111
// ASSGN SYS006,3330,VOL=DOS111,SHR
// EXEC IDCAMS,SIZE=30K
      BLDINDEX INFILE(DD1) OUTFILE(DD2)

      DEFINE PATH -
        (NAME(PL1VSAM.AJC2.NUMPATH) -
         PATHENTRY(PL1VSAM.AJC2.NUMIND))
/*
/&
```

Figure 73. Creating an Alternate Index Path for a KSDS

Using a Unique Key Alternate Index Path with a KSDS

Figure 74 on page 198 shows the use of a path with a unique key alternate index to update a KSDS and then to access and print it in the order of the alternate index.

The alternate index path is associated with the PL/I file by a DLBL statement that specifies the name of the path (given in the DEFINE PATH command in Figure 73) as the data set name (file identifier).

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.

```
// JOB DOS10#16 A.J.COX,N081,H205434
// OPTION LINK
// EXEC PLIOPT, SIZE=256K
    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);
          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;
/*
// EXEC LNKEDT
// DLBL NUMFLE1,'PL1VSAM.AJC2.NUMPATH',,VSAM
// EXTENT SYS006,DOS111
// DLBL NUMFLE2,'PL1VSAM.AJC2.NUMPATH',,VSAM
// EXTENT SYS006,DOS111
// ASSGN SYS006,3330,VOL=DOS111,SHR
// EXEC ,SIZE=256K
RIERA L             123
/*
// EXEC IDCAMS,SIZE=30K
    DELETE -
         PL1VSAM.AJC2.BASE
/*
/&
```

Figure 74. Using a Unique Alternate Index Path to Access a KSDS

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 Chapter 9. They use the same telephone directory data, but use the number as the key to the record. The record contains only the name.

```
// JOB DOS10#17 A.J. COX,N081,H205434
// OPTION LINK
// EXEC IDCAMS,SIZE=30K
         DEFINE CLUSTER -
             (NAME(PL1VSAM.AJC3.BASE) -
             VOLUMES(DOS111) -
             NUMBERED -
             TRACKS(2 2) -
             RECORDSIZE(20 20))
/*
// EXEC PLIOPT, SIZE=64K
 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));
         IOFIELD=NAME;
         WRITE FILE(NOS) FROM(IOFIELD) KEYFROM(NUMBER);
         GO TO NEXT;
 FINISH: CLOSE FILE(NOS);
 END CRR1;
/*
// EXEC LNKEDT
// DLBL NOS,'PL1VSAM.AJC3.BASE',,VSAM
// EXTENT SYS006,DOS111
// ASSGN SYS006,3330,VOL=DOS111,SHR
// EXEC ,SIZE=64K
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 75. Defining and Loading a Relative-Record Data Set (RRDS)

Defining and Loading a Relative Record Data Set

In Figure 75, the data set is defined with a DEFINE CLUSTER command and given the name PL1VSAM.AJC3.BASE. The fact that it is to be 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.

```
// JOB DOS10#18 A.J.COS,N081,H205434
// OPTION LINK
// EXEC PLIOPT, SIZE=64K
 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;
           PUT FILE(SYSPRINT) SKIP EDIT
               ('DUPLICATE',NAME)(A(15),A);
         END;
 NEXT:   GET FILE(SYSIN) EDIT(NAME,NEWNO,OLDNO,CODE)
                   (A(20),2 A(2),X(55),A(1));
             IF CODE='A' THEN GO TO RITE;
               ELSE IF CODE='C' THEN
             DO;
              DELETE FILE(NOS) KEY(OLDNO);
              GO TO RITE;
             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;
 RITE:   WRITE FILE(NOS) KEYFROM(NEWNO)
                       FROM(NAME);
                 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;
/*
// EXEC LNKEDT
// DLBL NOS,'PL1VSAM.AJC3.BASE',,VSAM
// EXTENT SYS006,DOS111
// ASSGN SYS006,3330,VOL=DOS111,SHR
// EXEC ,SIZE=64K
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
/*
// EXEC IDCAMS,SIZE=30K
      DELETE -
          PL1VSAM.AJC3.BASE
/*
/&
```

Figure 76. Updating an RRDS

The PL/I file is associated with the data set by the DLBL statement that uses as the data set name (file identifier) the name given in the DEFINE CLUSTER command.

Updating a Relative-Record Data Set

Figure 76 on page 200 shows an RRDS being updated. A DIRECT UPDATE file is used and new records are written by key. There is no need to check for the records being empty because the empty records are not available under VSAM.

In the second half of the program, starting at the label PRINT, the updated file is printed out. Again there is no need to check for the empty records as there is in REGIONAL(1).

The PL/I file is associated with the data sets by a DLBL statement that specifies the data set name (file identifier) PL1VSAM.AJC3.BASE, the name given in the DEFINE CLUSTER command in Figure 75.

At the end of the example, the DELETE command is used to delete the data set.

CHAPTER 11. 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 design and 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.

CONVERSATIONAL PROGRAM CHECKOUT

The compiler can be used in conversational mode when writing and testing programs at a terminal. The conversational features are available to users where CMS facilities are present. The conversational facilities enable you to develop a PL/I program at a terminal, through which you can examine diagnostic messages for compilation and results and diagnostic messages for compilation and execution. 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 batch processing.

The PL/I program is entered and processed using the DOSPLI, EDIT, and other CMS commands and features, described in DOS 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 listing and associated compiler options are discussed in Chapter 4. The diagnostic messages produced by the optimizing compiler are identified by a number prefixed by the letters IEL. These diagnostic messages are designed to be as self-explanatory as possible. Each message is reproduced in DOS PL/I Optimizing Compiler: Messages, which 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, even if the message indicates 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 an erroneous statement in the source program, can introduce a further, perhaps most 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 tables and cross-reference tables. The attribute table, specified by the ATTRIBUTES option, is used 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 statement 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 NOSYNTAX, NOCOMPILE, and NOLINK options. These options, when specified, will suppress subsequent 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 the 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 the compilation to be suppressed after the syntax-checking stage if syntax checking causes the compiler to issue diagnostic messages at or above the severity level specified in the option.

The NOLINK option specified with the severity level "W," "E," or "S" will cause link-editing (and execution) to be suppressed if the compiler issues 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 relocatable subroutine modules has been performed correctly. Diagnostic messages produced by the linkage editor are prefixed by "21." These messages are fully documented in DOS: Operator Communications and 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 executable program phase. The module map names the relocatable object 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 an executable program phase. The linkage editor is described in Chapter 5.

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 listed in DOS PL/I Optimizing Compiler: Messages. These messages are identified by the prefix IBM. The messages are printed on the SYSPRINT file if it is declared explicitly or implicitly as a PRINT file with a line size of not less than 72 or if the SNAP option occurs in the same program. Otherwise, the messages are printed on the operator console.

A failure in the execution of a PL/I program could be caused by one of the following:

- Logical error in source program
- 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 misuse of array subscripts, misuse of locator variables, failure to convert correctly from arithmetic to string data and from string to 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, will result in a 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 or of pointer variables, conversion errors, incorrect arithmetic operations, or string manipulation operations can cause this type of failure. See the <u>OS and DOS PL/I Language Reference Manual</u> for descriptions of other common errors that are made in PL/I source programs.

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 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;
```

You are advised to include such an on-unit in any PL/I program under test. The PUT DATA statement causes the printing of the names and the values of all identifiers valid for data-directed transmission and known in the block containing the on-unit in which it appears. Note that the use of the "PUT DATA;" statement requires a considerable amount of storage for symbol tables and library subroutines, some of which may never be used. Therefore the "PUT DATA;" statement is not recommended for permanent inclusion in a PL/I program, but only as a temporary aid to program checkout. The statement "ON ERROR SYSTEM;" contained in the on-unit ensures that further errors caused by uninitialized variables do not result in a permanent loop. Note also that if the program contains nested procedures or begin blocks, the on-unit shown above should appear in each block so that variables known only with a particular block can be printed.

A common error in programs that are incompletely checked-out is the failure to initialize variables before their values are used.

It is recommended that you not use the full optimization facilities of the compiler in the early stages of debugging because of the additional compilation required to provide the additional optional optimization. Full optimization can also cause the reordering of some expressions and statements. This may make debugging more difficult for programs that contain errors.

Insufficient Storage

It is possible to receive a diagnostic message indicating that there is insufficient address space available for a PL/I program or program phase, even if you are certain that there is enough storage for the job. VSE job control allocates space based on the size of the largest program in the library whose name begins with the same first four letters as the program or phase awaiting execution. Thus, if your program or phase has a name whose first four letters are the same as those in the name of a larger program in the library, job control will attempt to acquire the larger amount of space for your program. The solution is to rename your program or phase with a name whose first four letters are unique. Job control will then take the size from your program itself.

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. The VERIFY built-in function can be used to check the validity of input.

Unidentified Program Failure

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 prevented 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 partition that contain executable instructions, particularly the communications region, which contains the address tables for the execution-time error handling routines. The types of PL/I source program errors that might cause the main storage to be overwritten erroneously are:

1. Assignment of a value to a nonexistent array element. For example:

   ```
   DCL ARRAY(10);
    .
    .
    .
   DO I = 1 TO 100;
   ARRAY(I) = VALUE;
   END;
   ```

   To detect this type of error at execution time, enable the SUBSCRIPTRANGE condition. For each attempt to write a value to an element outside the declared range of subscript values, the SUBSCRIPTRANGE will be raised. If there is no on-unit for this condition, a diagnostic message will be printed. This facility, although a valuable program-checkout aid, is expensive in execution time and storage space, and should be used carefully during the debugging of programs containing arrays.

2. 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 a record-oriented transmission statement. 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 based variable when its qualifying pointer value has been changed. For example:

   ```
   DCL A STATIC,B BASED (P),Q POINTER;
   ALLOCATE B;
   Q = ADDR(A);
   P = Q;
   FREE B;
   ```

3. The use of incorrect values for label, entry, and file variables. Errors similar to those described above for locator variables are possible for label, entry, and file values that are transmitted and subsequently retrieved.

4. 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';
   ```

   Enabling STRINGRANGE will detect this. STRINGRANGE, although a valuable program checkout aid, is expensive in execution time and storage space, and should be used carefully during the debugging of programs containing the SUBSTR pseudovariable.

   The SIZE condition should be enabled when debugging programs so that any instances of the loss of high order bits or digits in an assignment, intermediate result, or input/output operation are detected.

Compiler or Library Subroutine Failure

If you are convinced that the failure is caused by a compiler 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 F on page 326.

Meanwhile, you can attempt to find an alternative way to perform the operation that is causing the trouble. A bypass is often feasible, because the PL/I language frequently provides alternative methods 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. If execution of a PL/I program results in a diagnostic message indicating an illegal supervisor call (SVC), it is probable that the DOS control program was generated without the particular SVC required by a PL/I function. Operating System requirements are given in DOS PL/I Optimizing Compiler: Installation.

STATEMENT NUMBERS AND TRACING

Three compiler options provide valuable program-checkout aids. They are the statement numbering option (GOSTMT), the statement number trace facility (FLOW(n,m)), and the statement count option (COUNT).

The GOSTMT option causes the number of the statement containing an error that results in the abnormal termination of execution to be printed as part of the execution-time diagnostic message. This option is not specified by default, so it must be given explicitly for each compilation.

The (FLOW(n,m)) option produces a list of the numbers of the last 'n' branch-out/branch-in statement numbers, and the last 'm' procedures and on-units to be entered. A branch-out statement is a statement such as a GOTO statement that transfers control to a statement other than the one that immediately follows it. A branch-in statement is a statement such as PROCEDURE, ENTRY, or any other labeled statement that receives control from a statement other than the statement that immediately precedes it. The figure you choose for "n" should be large enough to provide a usable trace of the flow of control through the program. The trace is printed whenever an on-unit with the SNAP option is entered. It gives both the statement numbers and the names of the containing procedures, blocks, or on-units. For example, an ERROR on-unit that causes 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;
```

The COUNT option produces a table indicating how many times each statement or group of statements in the program has been executed. Count output is written on the file SYSLST when the program terminates. The output has the following format:

```
PROCEDURE name1
     FROM     TO      COUNT
        1     20          1
       21     30         10
        .      .          .
        .      .          .
      200    210          i

 name1     TOTAL          m

PROCEDURE name 2
     FROM     TO       COUNT
        1     10           5
        .      .           .
        .      .           .
 name2     TOTAL           n
```

Three such columns are printed per page. Each column is followed by the total count for the column (that is, m+n in the preceding example.)

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

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 often 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 either PUT DATA statements or the CHECK prefix option.

A CHECK prefix option can specify program variables in a list. Whenever a variable that has been included in a checklist is assigned a new value, the CHECK condition is raised. If the implicit action for the CHECK condition is used, the action 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 options list is:

```
  (CHECK(A,B,C,L)):/* CHECKOUT PREFIX LIST */
    TEST:  PROCEDURE OPTIONS(MAIN);
         DECLARE A etc.,
              .
              .
              .
```

CHECK can also be used to print the name of a label constant whenever control passes through it.

If the CHECK condition is to be raised for all the variables and labels used in a program, the CHECK prefix option can be more simply specified without a list of items. For example:

```
  (CHECK):  TEST:   PROCEDURE;
```

Note that, if a CHECK on-unit, other than on CHECK SYSTEM, and a CHECK prefix-list are used, the implicit action (to print the name and value of the identifier that raises the CHECK condition) will be overridden. Although a valuable program-checkout aid, the CHECK condition is expensive in execution time and storage space.

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 to 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 a condition that raises the ERROR condition when there is no on-unit is to raise the FINISH condition and terminate the program. A dump of main storage is also produced if the DUMP option of the OPTION job control statement has been specified (see "Dumps" on page 209).

The FINISH condition is also raised for the following:

- When a SIGNAL FINISH statement is executed.
- When a PL/I program completes executing normally.

- When an ERROR on-unit is completed that does not return control to the PL/I program by means of a GO TO statement.
- When a STOP statement is executed.

The SIGNAL FINISH statement does not terminate the execution of a program.

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 to subsequently remove them from the program when it is found to be satisfactory. The PL/I preprocessor can be used to facilitate the inclusion of a standard set of 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 is compiled for execution without them.

A standard set of PL/I program checkout statements would require the enabling of any conditions that are disabled by default. The %INCLUDE statement that causes the inclusion of the set of program checkout statements should be placed at the start of the program. If there are any on-units that must remain in the program permanently and for which there are equivalent on-units in the included statements, the source statement book should precede any of the permanent on-units in the text of the source program.

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 condition built-in function ONCODE in the on-unit. The condition codes are described in the OS and DOS PL/I Language Reference Manual.

DUMPS

The checks given above will very rarely fail to reveal the cause of the error. In the unlikely event that an error cannot be detected by any of these methods, or in exceptional circumstances, when a compiler fault is suspected, it may be necessary to obtain a printout, or dump, of the main storage partition used by the program. A dump can display the contents of all buffers associated with PL/I files, the Pl/I file attributes for each file open when the dump is taken, and a trace of the block invocations that occurred during execution before the dump was taken. A hexadecimal dump can be obtained to determine the machine instructions and data present in main storage when the failure occurred. A report of the storage currently in use can be obtained to help in estimating the amount of storage required to execute the program. Refer to DOS PL/I Optimizing Compiler: Execution Logic for information about the organization of the object programs produced by the optimizing compiler, and how to interpret a storage dump. A map of the offsets of static and automatic variables from their defining bases is given if the MAP compiler option is used. This will enable you to identify variables in a dump.

A formatted PL/I dump is obtained by a call to PLIDUMP. The symbolic device SYSLST must be assigned to a printer or tape unit when the dump is taken.

PLIDUMP can be called with two optional arguments. The first argument is a character-string constant used to specify the types of information to be included in the dump. The second argument is a character-string expression with which you can identify the output produced by PLIDUMP. The format of the PLIDUMP statement is:

```
  CALL PLIDUMP [('option -list',
                ['user-identification'])];
```

The option-list is a contiguous string of characters chosen from among 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.

C To request continuation of the program after completion of the dump.

H To request a hexadecimal dump of the main storage partition used by the program.

NH To suppress the hexadecimal dump.

B If T is specified, B produces a separate hexadecimal dump of control blocks such as the TCA (task communications area) and the DSA (dynamic storage area) chain that are used in the trace analysis. If F is specified, B produces a separate hexadecimal dump of control blocks used in the file analysis, such as the FCB (file control block).

NB Suppresses hexadecimal dumps of control blocks.

D To request analysis of the status of any files that are open, and to provide useful debugging information regarding modules associated with the file.

ND To suppress the output produced by D above.

48 To request that the translate table from hexadecimal to character contains the PL/I 48-character set only.

60 To request that the translate table from hexadecimal to character contains the PL/I 60-character set.

R To request a report of the current storage in use and the amount of unused storage.

NR To suppress the output produced by R above.

Q To request a DOS system dump of the current partition using a minimum amount of storage (as opposed to normal PLIDUMP requirements). This option is intended only as a 'last resort option', since no extra information, apart from the hexadecimal dump, is given. The option can only be used if all options using more storage than the PLIDUMP dump routines are negated. Thus:

```
CALL PLIDUMP('NTNFNRQ');
```

must be coded to obtain this facility.

NQ To suppress the output produced by Q above.

The defaults for the above options are:

```
CALL PLIDUMP('TFRC48D');
```

The user-identification permits you to specify a character-string expression to identify individual dumps. It can only be specified if preceded by the specification of an options list, thus:

```
CALL PLIDUMP('TFHB',
'THIS IS MY OWN TRUE DUMP');
```

If the TSTAMP option was specified during compiler installation, the time and date of compilation will be held in the static internal control section for each procedure. The offset will be held in the first word of the control section. The information will take the form "day month year hour minute second": for example, 4 JULY 76 06.06.57. This will be printed at the head of the static storage by PLIDUMP if the B (block) option is in effect.

The static internal control section is addressed by register 3 in compiled code.

Because the ERROR condition is occasionally raised by errors about which little or no information can be printed out, the PLIDUMP routine is called automatically as part of the implicit action for ERROR if the DUMP option of the OPTION job control statement is specified for the job or job step. PLIDUMP is called with the option-list 'HB', and the dump is printed on SYSLST.

The PLIDUMP produced by implicit action is obtained by calling a special ON FINISH unit which then calls PLIDUMP with the options:

```
CALL PLIDUMP ('HB','ERROR ACTION');
```

This allows on-unit information to be given in the PLIDUMP. The PLIDUMP will contain irrelevant information in the first line of the trace. This line states:

```
'PLIDUMP was called from offset x from
an ON FINISH unit.'
```

This line can safely be ignored. Note that ON file information will only be given if an on-unit has been entered for that condition.

Trace Information

Trace information produced by PLIDUMP includes for procedures the name and offset within the program, and for begin blocks the offset only. When the statement number option GOSTMT is used, the dump includes the number of the statement that invokes each block. For on-units, the dump reproduces the values of any oncode built-in functions that could be used in the on-unit, regardless of whether the on-unit actually used the oncode 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 IBMDERR
- The PSW address or the address from which IBMDERR 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 DOS PL/I Optimizing Compiler: Execution Logic.

If the FLOW option was specified for the compilation, a table of statement numbers indicating the flow of control through the program is produced.

File Information

File information produced by the PLIDUMP modules includes:

1. Default and declared attributes of all files that are open
2. Contents of all buffers that are accessible to the dump routine.

The above information is given in BCD notation.

If the Block ('B') or the Hexadecimal ('H') option is requested, then according to the options:

3. Address and contents of FCB (file control block) are given.
4. Address and contents of ENVIRONMENT block (if any) are given.
5. Address and contents of DTF (define-the-file) block and ACB (address control block) are given.
6. For VSAM, the address and contents of the IOCB and appendage are given.

Because of the many variations in control block length, the length given for any one of items 2, 4, 5, and 6 cannot be guaranteed to be the exact length of that block. Usually, too large a block will be printed, but in some cases too small a block will be printed. If too small a block is printed, it may be necessary to use the full hexadecimal dump to obtain the complete file contents, using the link-edit map.

Debugging Information

The debugging option ('D') applies to the file information and provides extra information regarding the status of the file by decoding certain key flags within the control blocks. The names of the transmitter and open modules are also provided.

REPORT Option

The report produced gives the size and addresses of the following:

1. Partition
2. Problem program
3. Program management area
4. Primary LIFO storage area
5. Primary non-LIFO storage area. Within the primary non-LIFO storage:
    a. Free areas
    b. LIFO overflow segments
    c. Transmitter areas
6. Total storage used
7. Total storage unused

Note: A sample report is given in the <u>DOS PL/I Optimizing Compiler: Execution Logic</u>.

QUICK DUMP Option

The quick dump option (option 'Q') gives a DOS system dump from the beginning of the partition to its end (the PDUMP macro is used to obtain the dump). The quick dump option is only applicable if all other options are negated.

Hexadecimal Dump

The hexadecimal dump is a dump of the partition of main storage containing the program. The dump is given as three columns. The two left-hand columns contain the contents of storage in hexadecimal notation. The third column contains a BDC translation of the first two columns. For hexadecimal characters that cannot be represented by a BDC character, a period is printed. The hexadecimal characters are converted to characters using the 48- or 60-character set, depending on which option is specified.

EXECUTION-TIME RETURN CODES

A PL/I program invoked by an assembler language routine will return a code in register 15.

The return code can be set by passing as an argument to the CALL PLIRETC statement a value represented as a constant or a variable with the attribute FIXED BINARY (31,0): for example, CALL PLIRETC(12);. The range of codes should be restricted to 1 through 999. If a return code of greater than 999 is specified, the return code is set to 999 and a diagnostic message is issued. Codes that use values in the thousands are returned if an error causes the program to terminate.

The return code generated by a PL/I program consists of two 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. In the resulting total, the thousands digit indicates the way in which your program terminated. The hundreds, tens, and units are zero by default or set by your program when PLIRETC is called; they can be used to allow conditional execution of the next step or for any other purpose you require.

The meaning of the codes 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 option.
2000	ERROR condition raised and program terminated without return from ERROR or FINISH on-unit.
3000	Return codes in the 3000-3999 range can be issued by a user-written IBMBEER.
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.
4008	Code returned if PL/I program has no main procedure.
4012	Not enough main storage available.

4028 Excessive fragmentation of storage. PL/I's maximum working storage is 255 fragments.

4032 The DSA chain fields have been overlaid.

If a return code in the 4000-4032 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 F on page 326 describes the materials that will be required for examination by IBM in such circumstances.

CHAPTER 12. LINKING 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 these conventions:

- PL/I parameter passing conventions: These are adopted by PL/I to allow the length of nonarithmetic data items to be passed automatically to a called routine.
- 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 character string being passed directly, rather than the address of a control block that contains the length and address of the character string.

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 "Arguments, Parameters, and Return Codes" on page 226.)

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 STXIT PC 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 that is chained in the normal way. When PL/I calls an assembler subroutine, the subroutine must either forego the use of register 12 or cancel and reissue the STXIT PC macro instruction, thus 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 STXIT PC 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, if they do, 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 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 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 from 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 a PL/I routine calls an assembler-language routine that in turn calls a PL/I routine, the PL/I routines must be link-edited together. For example, if an assembler-language program loads a PL/I routine, the results are unpredictable.

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 want to make consistent use of assembler-PL/I programming, you should, however, read the remaining sections of this chapter to understand the reasoning behind the code.

PL/I CALLING ASSEMBLER SUBROUTINES: Unless you have good reasons for wanting to do your own error-handling, you should use the code in Figure 77 on page 218 for a nonrecursive routine and the code in Figure 79 on page 222 for a recursive or reentrant routine. If the routine is to receive parameters or to return values, study "Arguments, Parameters, and Return Codes" on page 226 at the end of this section.

ASSEMBLER CALLING PL/I SUBROUTINES: If your PL/I subroutine is invoked only once, it should, if possible, be given the MAIN option and called via entry point PLICALLA as in the first line of Figure 79, ending at EOJ. If it 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 82 on page 225 and then call PLICALLA.

If your routine is to be called a number of times, you should follow the complete scheme shown in Figure 79. If parameters are to be passed or values returned, study the section "Arguments, Parameters, and Return Codes" on page 226.

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 DOS 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 IBM linkage conventions, which include the storing of register values in a save area, and by ensuring that the content of register 12 is not altered by the assembler routine if PL/I is to handle interrupts that occur during execution of the assembler routine. (It is sensible to allow PL/I to handle interrupts. The alternative involves resetting the program interrupt exit twice and is a considerable overhead. The disadvantage of using PL/I error-handling is that it prevents you from using register 12 at any time during the assembler program.) Register 13 must be set to the address of a new save area established by the assembler routine.

If you intend to call assembler language subroutines from PL/I, you need to know no more about the PL/I environment and should continue with "Calling Assembler Routines from PL/I" on page 218.

ESTABLISHING THE PL/I ENVIRONMENT

An assembler-language routine that invokes a PL/I procedure for which the PL/I environment has not been established can use one of two entry points to establish the environment. The entry points are given the standard names PLICALLA and PLICALLB, and are described later in this section.

Use of PLIMAIN to Invoke PL/I Procedure

After the environment has been created, an address held in a control section called PLIMAIN is used to transfer control to the PL/I procedure whose address is also contained in PLIMAIN. Normally, after link-editing, PLIMAIN will contain the entry-point address of the first, or only, PL/I main procedure in the program. If the assembler-language routine is to invoke a PL/I procedure that is not the first, or only, main PL/I procedure in the program, it must insert in the compiler-generated control section PLIMAIN the address of the entry-point of the procedure it is to invoke. The example in Figure 82 on page 225 shows how this is done.

If there is no main procedure in the program, the assembler routine should contain an entry point called PLIMAIN, at which is held the address of the entry point of the PL/I routine to be invoked. The example in Figure 83 on page 225 shows how the appropriate address is inserted into the location represented by the entry point PLIMAIN. If the assembler program does not include an entry point called PLIMAIN in these circumstances, a dummy module called PLIMAIN will be included from the DOS PL/I Resident Library, thus incurring an avoidable overhead in time and space.

After the PL/I environment has been established, it can, as shown in the example in Figure 79 on page 222, be preserved, and any PL/I procedure can be invoked subsequently by loading the address of its entry point into a register, and executing a branch-and-link-register instruction to it.

PLICALLA AND PLICALLB

PLICALLA: PLICALLA is the entry point to be used when the PL/I environment must be established for a PL/I procedure that can use for its dynamic storage as much of the available space in storage as it requires.

PLICALLB: PLICALLB is the entry point to be used when the PL/I environment must be established for a PL/I procedure that can use for its dynamic storage only a specific amount of the available storage at a specified address.

Further details and examples using PLICALLA and PLICALLB are given later in this chapter.

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 automatic variables and miscellaneous housekeeping items.

An assembler routine invoked from PL/I must take one of the following actions to preserve the PL/I environment:

1. On invocation, 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. 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 not to use a PL/I error-handler and does not invoke a function routine, the STXIT PC macro must be used to reset the interrupt handler, but only those registers that it modifies need be stored. The STXIT PC macro is discussed later in this chapter.

CALLING ASSEMBLER ROUTINES FROM PL/I

The following section describes:

- How to invoke a non-recursive assembler routine.
- How to invoke a recursive assembler routine.

INVOKING A NONRECURSIVE ASSEMBLER ROUTINE

When a PL/I program invokes a nonrecursive assembler-language routine, the assembler-language routine must follow System/370 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 within the assembler routine itself. Prior to 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. The assembler instructions in Figure 77 should be executed immediately when the assembler routine is invoked in order to achieve the given objectives. The example assumes that the assembler routine uses register 10 as its base register.

```
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    13,4(4)            STORE PL/I DSA ADDRESS IN SAVE AREA
         ST    4,8(13)
         LR    13,4               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 77. Skeletal Code for a Nonrecursive Assembler Routine to be Invoked from PL/I

If you use the code in Figure 77 around your assembler program, you will be able to assemble and link-edit it, and then call it with a PL/I CALL statement in a perfectly straightforward manner.

INVOKING A RECURSIVE 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 as illustrated in Figure 77. The suggested method 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 want to use PL/I error-handling in the assembler routine.

If you do not want to use Pl/I error-handling in your subroutine, you must set the first byte of the DSA to X'00'. (You must also issue a STXIT PC macro to disable and enable the PL/I error-handler at the start and end of the routine. See "Overriding and Restoring PL/I Error-Handling" on page 226.) The DSA obtained must be at least 80 bytes long. Additional storage can be obtained for use in the assembler routine. The total length of storage obtained must be a multiple of 8 bytes.

If PL/I error-handling is to be retained in the assembler language routine and the assembler routine is not, in turn, going to call PL/I subroutines, the DSA should be at least 88 bytes in length, byte 0 must be set to X'80', byte 1 to X'00', and bytes 86 and 87 (the PL/I error-handler enabler cells) set to X'91C0'. Additional storage can be obtained within the DSA for use by each invocation of the assembler subroutine. The total length of the DSA must be a multiple of 8 bytes.

Also, the entry point should be preceded by the name and length 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 the one byte length field (containing the length of the name in hex), which immediately precedes the entry point of the assembler routine.

The example in Figure 78 on page 220 shows how to create and release a DSA in a recursive assembler routine. The contents of registers 12 and 13 and the layout of storage in a recursive environment are described in _DOS PL/I Optimizing Compiler: Execution Logic_.

USE OF REGISTER 12

An assembler routine that is to be invoked by a PL/I procedure should not modify register 12, because the value in this register will be changed by the PL/I error-handling routines if a program check interrupt occurs in the assembler routine.

If PL/I error-handling is not required, the assembler routine should issue a DOS Supervisor STXIT PC macro to establish either its own or the system error-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 226. (A routine that changes the contents of register 12 should also store it on entry and restore it on return.)

```
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 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 ADDRESS.
          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 OVER-
          BALR  14,15          FLOW 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'
          MVI   86(13),X'91'   PRESERVE PL/I 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
```

Note: If your assembler routine requires separate storage for each invocation, it should be added to the value in the load address instruction (LA 0,88) and addressed from register 13. Total length must be a multiple of 8.

Figure 78. Skeletal Code for a Recursive Assembler Routine that Uses the PL/I Storage Scheme

CALLING IOCS MODULES FROM ASSEMBLER SUBROUTINES

If a subset of an IOCS module is called by a user's assembler subroutine, and a different subset of the same module is called by PL/I, then duplicate entry points may be diagnosed at link-edit time. This can be resolved by replacing the two LIOCS modules in the relocatable library by their common superset module.

A list of the IOCS modules subject to duplicate entry point messages can be found in the DOS PL/I Optimizing Compiler: Installation Guide.

The appropriate supersets can be coded from the publication VSE/Advanced Functions Macro Reference, or equivalent publication.

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 branch and to link to it. When PLICALLA is used in this way, the PL/I environment is created and control is then passed via PLIMAIN to the first (or only) main procedure in the program. Use of this technique will cause the PL/I environment to be established separately for each invocation.

Note: PL/I procedures may not be loaded into a GETVIS area.

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 79 on page 222 contains an assembler-language routine that establishes the PL/I environment once only for multiple invocations of PL/I procedures.

In Figure 79, the assembler routine MYPROG receives control initially from the supervisor, and invokes the PL/I 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. At this entry point, the PL/I environment is stored, and the new DSA, 100 bytes in length, is created in a manner similar to that previously given for creating a DSA in a recursive routine. If there is insufficient room for the new DSA, the PL/I overflow routine is invoked to attempt to obtain storage elsewhere for the new DSA. The overflow routine is described in greater detail in _DOS PL/I Optimizing Compiler: Execution Logic_.

In the assembler routine, the instructions following the label ENOUGH, through the instruction that loads the address of the PL/I entry point HEAD, are all concerned with setting up the DSA so that the correct environment exists when the routine invokes the external PL/I procedures PLIN and PLOUT or 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 (PLIN in this example) that returns a value will assign that value to the last address in the parameter list PARMLST1. This address is the address of the assembler-defined storage for RESULT. The constant X'80' in the first byte of the fullword containing the address of RESULT in PARMLST1 indicates that it is the last fullword in the parameter 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, no value is returned to the invoking routine. Similarly, the procedure PLOUT contains the parameterless entry point FOOT and does not return a value.

The INCLUDE statement in the link-edited step ensures that the resident library module IBMBPJRA is included in the executable program. IBMBPJRA contains, for the DOS implementation of the optimizing compiler, the entry points PLICALLA and PLICALLB.

```
// JOB FIG1203
// OPTION LINK,DUMP
// EXEC ASSEMBLY,SIZE=64K
MYPROG   CSECT
         ENTRY ASSEM
         BALR  10,0                         ESTABLISH ADDRESSABILITY
         USING *,10
         LA    13,SAVEAREA
         SR    1,1
         L     15,=V(PLICALLA)              CALL THE PL/I PROCEDURE WHICH
         BALR  14,15                        -HAS OPTIONS(MAIN) AND SO SET
*                                           -UP THE PL/I ENVIRONMENT AND
*                                           -THEN CALL ASSEM.
*
         LTR   R15,R15                      NORMAL RETURN?
         BNZ   CANCEL                       NO
         EOJ
CANCEL   EQU   *
         CANCEL
*
*
         DC   C'ASSEM'                      THE NAME IN PL/I FORMAT
         DC   AL1(5)
ASSEM    DS   0H
         STM  14,12,12(13)                  STORE PL/I REGISTERS FOR
*                                           PROCEDURE 'MAIN'.
*
         BALR  10,0                         ESTABLISH ADDRESSABILITY
         USING *,10
*                                           GET STORAGE FOR A SAVE AREA
         LA    0,104                        LENGTH REQUIRED (104 BYTES)
         L     1,76(13)                     ADDRESS OF START OF CURRENTLY
*                                           AVAILABLE STORAGE.
         ALR   0,1
         CL    0,12(12)                     IS THERE ENOUGH SPACE LEFT?
         BNH   ENOUGH                       YES
         L     15,116(12)                   LOAD ADDRESS OF OVERFLOW
                                            -ROUTINE AND BRANCH TO IT.
         BALR  14,15
ENOUGH   EQU   *
         ST    0,76(1)                      STORE ADDRESS OF START OF
*                                           -REMAINING AVAILABLE STORAGE
                                            -IN NEW DSA OFFSET 76.
*
         ST    13,4(1)                      STORE CHAIN BACK ADDRESS
         ST    1,8(13)                      STORE CHAIN FORWARD ADDRESS
         MVC   72(4,1),72(13)               COPY ADDRESS OF WORKSPACE FOR
*                                           -USE BY 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 ERROR
         MVI   86(13),X'91'                 -HANDLING IN THE ASSEMBLER
         MVI   87(13),X'C0'                 -ROUTINE.
*
         SR     5,5                         R5 MUST BE ZERO FOR CALLING AN
*                                           -EXTERNAL PL/I PROCEDURE
*
         SR     1,1                         R1 MUST BE SET TO ZERO FOR A
*                                           -PARAMETERLESS ENTRY POINT THAT
*                                           -DOES NOT RETURN A VALUE.
*
         L      15,=V(HEAD)                 CALL PL/I TO 'HEAD' PAGE
         BALR   14,15
*
```

Figure 79 (Part 1 of 2). Invoking PL/I Procedures from an Assembly-Language Routine

```
LOOP     EQU    *
         LA     1,ARGTLST1                    CALL PL/I TO READ AND ADD
         L      15,=V(PLIN)
         BALR   14,15
*
*
         L      3,RESULT                      TEST RESULT -
         LTR    3,3                           -BRANCH OUT IF IT IS NEGATIVE
         BM     OUTLOOP
*
         LA     1,ARGTLST2                    CALL PL/I TO TRANSMIT RESULT
         L      15,=V(PLOUT)
         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 MAIN PL/I PROCEDURE
         LM     14,12,12(13)                  WITH OPTIONS(MAIN).
         BR     14
*
ARGTLST1 DC     A(DATA)
ARGTLST2 DC     X'80'
         DC     AL3(RESULT)
DATA     DC     F'123'
RESULT   DC     F'0'
SAVEAREA DC     18F'0'
         END    MYPROG
/*
// EXEC PLIOPT,SIZE=64K
* 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;
/*
 INCLUDE IBMBPJRA
// EXEC LNKEDT
// EXEC, SIZE=64K
 50  77  123  234  345  456  -23  -100  -123  -234
/*
/&
```

Figure 79 (Part 2 of 2). Invoking PL/I Procedures from an Assembly-Language Routine

ESTABLISHING THE PL/I ENVIRONMENT SEPARATELY FOR EACH INVOCATION

If it is necessary to reestablish the PL/I environment each time a PL/I procedure is invoked, use the entry point PLICALLA (Figure 80) or PLICALLB (Figure 81) to invoke the PL/I initialization routines.

```
         LA     1,PARMLIST
         L      15,=V(PLICALLA)
         BALR   14,15
         -
         -
         -
         -
         -
PARMLIST DC     A(parm1)                    ADDRESS OF FIRST PARAMETER FOR PL/I
         DC     A(parm2)                    ADDRESS OF SECOND PARAMETER FOR PL/I
         -
         -
         -
         -
         DC     X'80'                       END OF PARAMETER LIST FLAG
         DC     AL3(parmn or return-value)  ADDRESS OF LAST PARAMETER
*                                           -OR RETURNED VALUE
```

Figure 80. Use of PLICALLA

```
         LA     1,PLIST
         L      15,=V(PLICALLB)
         BALR   14,15
         -
         -
         -
PLIST    DC     A(PARMLIST)                 ADDRESS OF PL/I PARAMETER LIST
         DC     A(LENGTH)                   LENGTH OF STORAGE FOR PL/I
*                                           -ON DOUBLE WORD BOUNDARY
         DC     X'80'
         DC     AL3(AREA)                   START OF PL/I STORAGE AREA
         DC     A(parm1)                    ADDRESS OF FIRST PARAMETER
         DC     A(parm2)                    ADDRESS OF SECOND PARAMETER
         -
         -
         DC     X'80'                       END OF PARAMETER LIST FLAG
         DC     AL3(parmn or return-value)  ADDRESS OF LAST PARAMETER
*                                           -OR RETURNED VALUE
LENGTH   DC     F'8192'                     ROUTINE'S STORAGE LIMITED TO 8K BYTES
AREA     DS    1024D                        ROUTINE'S STORAGE STARTS HERE
```

Figure 81. Use of PLICALLB

For PLICALLA, the assembler-language routine must insert in register 1 the address of the parameter list that contains the addresses of any arguments to be passed on to the PL/I procedure. For PLICALLB, the assembler-language routine must insert in register 1 the address of a parameter list (PLIST in Figure 81) that contains the following:

- The address of the parameter list containing addresses of arguments to be passed to PL/I,
- The address of the value for the amount of storage to be made available to the PL/I procedure, and

- The start address of the storage to be used by the PL/I procedure. The storage should be doubleword aligned. Note that the first byte in the last address word in each of these parameter lists must contain X'80'. The examples in Figure 80 on page 224 and Figure 81 show the use of PLICALLA and PLICALLB to invoke the first (or only) main PL/I procedure in the program.

If more than one PL/I subroutine is called and each subroutine must create and destroy the PL/I environment, the address of each subroutine, as it is required, should be placed in PLIMAIN before the call to PLICALLA or PLICALLB. The example in Figure 82 does this by setting the address in PLIMAIN to that of the external entry name MYPROG.

```
         LA      1,PARMLIST
         L       3,=V(PLIMAIN)                CHANGE ADDRESS IN PLIMAIN
         MVC     0(4,3)=V(MYPROG)             -TO THAT OF MYPROG
         L       *15,=V(PLICALLA)
         BALR    14,15
         -
         -
         -
         -
         -
         -
PARMLIST DC      A(parml                      FIRST PARAMETER TO MYPROG
         DC      X'80'
         DC      AL3(parm2)                   LAST PARAMETER TO MYPROG
```

Figure 82. Inserting a PL/I Entry Point Address in PLIMAIN

If it is necessary to reestablish the PL/I environment for each invocation of a PL/I procedure where there is no main PL/I procedure in the program, the use of either entry point PLICALLA or PLICALLB must be accompanied by the use of an entry point called PLIMAIN in the assembler-language routine. This entry point should contain the address of the PL/I routine to be invoked. Figure 83 shows how this is done.

```
         LA      1,PARMLIST
         L       2,=A(PLIMAIN)                INSERT ADDRESS IN PLIMAIN
         L       3,=V(MYPROG)                 -OF ENTRY TO MYPROG
         ST      3,0(2)
         L       15,=V(PLICALLA)
         BALR    14,15
         -
         -
         -
         -
         -
         -
PARMLIST DC      A(parml)FIRST PARAMETER TO MYPROG
         DC      X'80'
         DC      AL3parm2)LAST PARAMETER TO MYPROG
PLIMAIN  DS      F
```

Figure 83. Establishing PLIMAIN as an Entry in the Assembler-Language Routine
ENTRY PLIMAIN

PL/I Calling Assembler Calling PL/I

The information given in the preceding sections will be sufficient to write programs to include a PL/I procedure that invokes an assembler-language routine which invokes a further PL/I procedure. Figure 79 on page 222 contains an example of a program which performs this type of processing.

Assembler Calling PL/I Calling Assembler

The information given in the preceding sections will be sufficient to write programs that include an assembler-language routine that invokes a PL/I procedure which in turn invokes an assembler-language routine. Figure 79 contains an example of a program which performs this type of processing.

OVERRIDING AND RESTORING PL/I ERROR-HANDLING

The PL/I error-handling facilities are described in detail in DOS PL/I Optimizing Compiler: Execution Logic Manual. The following paragraphs explain how to override and restore these facilities in an assembler-language subroutine.

An assembler-language routine invoked from PL/I can override PL/I error-handling by issuing its own STXIT PC macro. However, a routine that issues a STXIT PC macro to cancel PL/I error-handling must restore the PL/I error-handling facilities before returning to the PL/I program. It does this by issuing a further STXIT PC macro before restoring the PL/I registers and branching back. The STXIT PC macro that restores the PL/I error-handling must have two address operands containing addresses of the PL/I error-handler and its save area. The example in Figure 84 shows how these two addresses are obtained.

```
PROGA  CSECT
       ENTRY   ASSEM                ENTRY POINT INVOKED FROM PL/I
       STM     14,12,12(13)         STORE PL/I ENVIRONMENT
       BALR    10,0                 ESTABLISH BASE REGISTER
       USING   *,10
       L       5,40(12)             OBTAIN ADDRESS AT OFFSET 40 FROM TCA
       LA      6,28(5)              OBTAIN ADDRESS OF THE PL/I ERROR-HANDLER
       ST      6,ADDR1              -AND STORE IT
       LA      6,36(5)              OBTAIN ADDRESS OF PL/I SAVE AREA FOR
       ST      6,ADDR2              -ERROR-HANDLING AND STORE IT
       STXIT   PC, (operands)       ESTABLISH NEW ERROR-HANDLER
       -
       -                            Assembler routine that modifies register
       -                            12 and either handles its own errors or
       -                            uses the system for error-handling
       LM       0,1,ADDR1           RESTORE OLD PL/I ERROR-HANDLER
       STXIT    PC,(0),(1)
       L        13,4(0,13)          RESTORE PL/I ENVIRONMENT
       LM       14,12,12(13)
       BR       14                  RETURN TO PL/I
ADDR1  DS       F                   STORAGE FOR PL/I ERROR-HANDLER ADDRESS
ADDR2  DS       F                   STORAGE FOR PL/I ERROR-HANDLER'S SAVE AREA AD
```

Figure 84. Methods of Overriding and Restoring PL/I Error-Handling

ARGUMENTS, PARAMETERS, 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 contains X'80' in its first byte to enable it to be recognized.

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.

The contents of the parameter list depend on whether OPTIONS(ASSEMBLER) was specified and whether the assembler subroutine is invoked by a function reference or a subroutine call. (Note that OPTIONS(ASSEMBLER) can only be used for assembler routines called from PL/I and not vice versa.)

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 _OS and DOS PL/I Language Reference Manual_.

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 of the parameter list is that of the byte that contains the start of the data item. Also, varying-length character 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.

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 the address either of a data item or 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 a "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 the _DOS PL/I Optimizing Compiler: Execution Logic Manual_.

It is recommended that the use of this type of linkage be avoided wherever possible. Access to locator descriptors is normally necessary only 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

In order to pass one or more arguments to a PL/I routine, an assembler routine must create a parameter list and set its address in register 1. The last fullword in the parameter list must have X'80' in its first byte. If the PL/I routine executes a RETURN(expression) statement, the last address of 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 _DOS PL/I Optimizing Compiler: Execution Logic Manual_.

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 twenty 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.

RETURN CODE

Assembler subroutines may pass a return code to a calling PL/I procedure by setting a value in the range 1 to 999 in the halfword location TURC in the TCA, that is, 70(0,12). The calling PL/I procedure can access this value by using the PLIRETV built-in function.

CHAPTER 13. CHECKPOINT/RESTART

Checkpoint/restart is a technique for resuming execution of a program after an interruption by, for example, a power failure. Checkpoint/restart involves the creation of checkpoint records that can be subsequently retrieved and used to restart the execution of the program. Execution is resumed at the point reached when the checkpoint record was made.

The use of checkpoint/restart should be confined to programs that are likely to run for long periods of time. Only programs in the background or a batched-job foreground partition may be checkpointed.

CHECKPOINTS

Checkpoint records can be created during execution of a PL/I program by including a CALL PLICKPT statement. Its use would normally be restricted to some natural point of progression in the program, such as between one set of calculations and the next.

Checkpoint records can be recorded on magnetic tape or on disk storage devices. The appropriate TLBL or DLBL and EXTENT statements must be supplied to define the checkpoint data set.

Checkpoint records cannot be written to SAM files in VSAM space.

RESTARTS

A program can resume execution from a checkpoint. The checkpoint record is loaded and control restored to it by a DOS facility which is specified by the RSTRT job control statement. The RSTRT statement is described in DOS/VSE System Control Statements. It may be useful to inform the operator that a particular program can be restarted if a failure occurs. The method used to pass this information will depend on local practice. The operator can then use the RSTRT statement directly to restart the program at the earliest opportunity.

PLICKPT

The format of the CALL PLICKPT statement is:

```
CALL PLICKPT[(arg1[,arg2[,arg3[,arg4]]])];
```

where

arg1 represents a character string that identifies the filename for a DLBL statement that defines the checkpoint data set on a disk storage volume. If the checkpoint record is to be written onto a magnetic tape, this argument should be a null string. If the checkpoint record is to be written onto a disk storage device and this argument is a null string, the filename SYSCHK is assumed, and should be used as the filename on the DLBL statement for the checkpoint data set.

arg2 represents a character string variable of at least four characters in length, which the DOS checkpoint macro will use to record the sequence number of the checkpoint record that is being recorded. The PL/I program should print this sequence number as soon as each checkpoint record has been made so that, if a restart is necessary, the latest checkpoint taken can be identified.

arg3	represents a character string that defines the characteristics of the device used to record the checkpoint records. The following format should be used for the character string:

```
[SYSnnn][,(2400|2311|2314
|3330|3340|3350|FBA)]
```

where SYSnnn is the symbolic device name of the device used for recording the checkpoint records. Symbolic device names in the range SYS000 through SYS255 can be used. Defaults for this argument are: SYS001 and 2400.

2311[1]	must be specified if the device used is a 2311 disk storage drive.
2314[1]	must be specified if the device used is a 2314 disk storage drive.
2400	must be specified if the device used is a magnetic tape drive.
3330[1]	must be specified if the device used is a 3330-1 disk storage drive. It may be specified if the device used is a 3330-11, but this is not required.
3340[1]	must be specified if the device used is a 3340 disk storage drive.
3350[1]	may be specified if the device used is a 3350 disk storage drive.
FBA[1]	may be specified if the device used is a fixed-block device.

[1]Programs compiled with 2311, 2314, 3330, 3350, or FBA can be used unchanged with other direct access storage devices, including 3375 and 3380. The user can override the devices in arg3 with the other direct access storage devices at execution time by means of the job control ASSGN statement.

arg4	represents a fixed binary variable of precision (31,0), which the DOS checkpoint/restart facilities will use to return a code to the PL/I program. The codes returned will have the following values and meanings:

0	Checkpoint successful. This code will be returned after a checkpoint record has been successfully made.
8	Checkpoint unsuccessful. This code will be returned if an argument is found to be invalid by the checkpoint routine. For example, the device type might be unrecognizable.
12	Checkpoint unsuccessful. This code will be returned if the checkpoint routine has encountered either a hardware error or an error in the set-up for taking checkpoints. Further details are given in the DOS System Programmer's Guide.
4	Restart has occurred. This code will be returned if a program has been restarted from a checkpoint.

All the arguments are optional. If arg2, arg3, or arg4 is specified, the defaults for preceding arguments will be assumed if these arguments are specified as null strings. For example, to specify arg4 only, code the PLICKPT statement as follows:

```
  CALL PLICKPT('','','',CODE);
```

where the variable CODE is to receive the return code set by the checkpoint/restart facilities.

TAKING CHECKPOINTS ON MAGNETIC TAPE

Any number of checkpoints can be recorded consecutively on magnetic tape. A separate magnetic-tape device can be used exclusively for the checkpoints. However, if there are no additional devices available, it is possible to include checkpoint records among the records of another data set that is being written onto magnetic tape; such records will be recognized by DOS data set management and ignored when the data set is accessed for input by a PL/I program (except when the data set is accessed by a SEQUENTIAL UNBUFFERED file). Similarly, records that are not checkpoint records will be ignored by the DOS restart routines when a checkpoint record on such a data set is used to restart execution of a program.

If a labeled magnetic tape is used exclusively for recording checkpoint records, no TLBL statement is required, and the MTC job control statement should be used to position the tape past the label.

TAKING CHECKPOINTS ON DISK STORAGE

Any number of checkpoints can be recorded on a disk storage volume. However, the number of checkpoint records that will be present on the volume at any one time depends on the amount of space made available to the checkpoint data set. When coding the EXTENT statement for the data set, the number of tracks required can be calculated as follows:

```
 tracks = (1+CEIL(P/T))*R

where P = partition size,
      R = number of checkpoint records
          to be retained, and
      T = track capacity
```

For fixed block devices, the storage requirements are specified in terms of fixed blocks. The storage requirements can be calculated as follows:

```
 blocks = (1+CEIL(P/B))*R

where B = hardware-dependent blocksize
          for the fixed block device.
```

The partition size must be that used for the execution of the PL/I program. When the extent is full, additional checkpoint records are written over the preceding checkpoints from the beginning of the extent.

If it is possible that the PL/I program will execute more than one CALL PLICKPT specifying a particular filename, the JCL for the job should include a DLBL statement specifying an expire time of zero for the corresponding data set. If this is not done, the operator will be called upon to delete the data set each time a succeeding version is to be written.

EFFECT OF RESTART ON DATA SETS

Any data sets used by a program that is to be restarted from a checkpoint must be available when the restart is attempted. Such data sets are repositioned before processing is resumed.

UNIT-RECORD DATA SETS

The repositioning of any unit-record data sets is the responsibility of the user. To facilitate the repositioning of these data sets, the following suggestions are given:

1. Checkpoints should be taken at a logical break in the processing of unit-record data sets.

2. Information displayed when a checkpoint is taken should identify the last record processed for each unit-record data set.

DATA SETS ON DIRECT-ACCESS DEVICES

Data sets residing on direct-access storage devices and being processed sequentially when the checkpoint was taken are repositioned by the restart routines.

Data sets residing on direct-access storage devices and being processed by a DIRECT UPDATE file present a problem in that any records updated after a checkpoint might be erroneously updated a second time when the program is restarted. There are two possible solutions to this problem:

- Design the program to maintain a history of all updates as a separate file. This file can then be used as a checklist against any erroneous updates.

- Set up the data set so that each record within it contains information that can be checked by the program to prevent a further erroneous updating.

DATA SETS ON MAGNETIC TAPE

Repositioning of magnetic tape volumes to the point following the last record to be processed is performed automatically provided the following points are observed. (The first three points involve the use of the MTC job control statement.)

1. If the data set is one of several on the volume, the volume should be positioned at the beginning of the relevant data set by means of a MTC statement.

2. Magnetic-tape volumes with nonstandard labels should be repositioned past the labels to a point (the tapemark) preceding the first record of the relevant data set.

3. If a BACKWARDS file is associated with a data set on a magnetic tape that is unlabeled or has nonstandard labels, the volume should be positioned immediately past the tapemark following the last record of the data set.

4. The correct volume of a multivolume data set should be mounted.

EXAMPLE

An example of a PL/I procedure that uses the checkpoint/restart facilities is given below. The example causes a checkpoint record to be written onto a 2311 disk storage device, for which it uses the symbolic device name SYS030, the data set name CKPT1, and the default file name SYSCHK.

```
// JOB CKPTRST
// OPTION LINK
// EXEC PLIOPT,SIZE=64K
 L1: PROC OPTIONS(MAIN);
     DCL RET_CODE FIXED BIN(31,0),
     COUNT CHAR(8);
     .
     .
     .
     /* STATEMENTS FOR CHECKPOINT/RESTART */

     CALL PLICKPT('',COUNT,'SYS030,2311',RET_CODE);
     /* THE PROGRAM CONTINUES OR RESTARTS AT THIS POINT */
     IF RET_CODE=0 THEN
            PUT EDIT('CHECKPOINT #',COUNT,'OK')(A);
     IF RET_CODE>4 THEN
            PUT EDIT('CHECKPOINT NOT TAKEN')(A);
     IF RET_CODE=4 THEN
            PUT EDIT('CHECKPOINT/RESTART HAS OCCURRED')(A);

     /*  END OF CHECKPOINT/RESTART STATEMENTS */
     .
     .
     .
     END L1;
/*
// EXEC LNKEDT
// ASSGN SYS030,X'294'
// DLBL SYSCHK,'CKPT1',0,SD
// EXTENT SYS030,DOS222,1,3458,60
//EXEC,SIZE=64K
/&
```

Figure 85. Example of PL/I Routine to Take Checkpoints

CHAPTER 14. PL/I SORT

If you intend to use the PL/I sort facilities, your installation must include a copy of the DOS/VS program product sort/merge program (5746-SM1), or a copy of the DOS program product sort/merge program (5746-SM2). The PL/I sorting facilities make use of the sort/merge program to arrange records according to a predetermined sequence. The sort/merge program includes user exit points to enable user-written routines to be entered at particular stages during the sorting operation and to provide access to records that are being sorted.

The PL/I sort facilities provide an interface to enable the sort program to be invoked and to call PL/I procedures through two of the user exits, E15 and E35. This chapter describes the method of invoking sort/merge from PL/I and the use of the user exits E15 and E35.

Storage Requirements

The minimum storage requirements for the sort program when used in conjunction with a PL/I program is 16K bytes. If a direct-access storage device is to be used for intermediate storage, at least 32K bytes of storage must be available.

Efficiency is enhanced if additional main storage can be provided. These storage requirements are in addition to those of the PL/I program.

ENTRY NAMES

A PL/I program invokes the sort program by means of a CALL statement that names one of four entry points to a PL/I sort interface routine provided by the DOS PL/I Resident Library. The CALL statement also passes arguments that specify the requirements for the sorting operation. The arguments include a sequence of sort/merge control statements in the form of character-string expressions. The PL/I sort interface module has entry points for four types of processing:

PLISRTA:	Invokes the sort/merge program to retrieve records from a data set (SORTIN), sort them, and write them in sorted sequence onto another data set (SORTOUT).
PLISRTB:	Invokes the sort/merge program and specifies the use of user exit E15. A PL/I procedure invoked at user exit E15 can supply all the records to be sorted. The sorted records are written directly onto the data set SORTOUT.
PLISRTC:	Invokes the sort/merge program and specifies the use of user exit E35. A PL/I procedure invoked at user exit E35 can receive all the records from the sort and handle any output that is required.
PLISRTD:	Invokes the sort/merge program and specifies the use of user exit E15 and user exit E35. The use of these user exits is exactly as described for PLISRTB and PLISRTC, above.

After completion of the sort, the sort/merge program returns a value to the invoking program to indicate whether the sort was successful. The invoking procedure must include a variable with the attributes FIXED BINARY(31) to receive this value, and the name of the variable must always be included in the argument list of the call statement that invokes sort/merge. The return-code values are:

0	Sort successful
16	Sort unsuccessful

PROCEDURES INVOKED VIA SORT USER EXITS

Both external and internal PL/I procedures can be invoked via sort user exits. The use of external PL/I procedures should present no problems so long as their entry names are properly declared in the main PL/I procedure and they are link-edited with the main PL/I procedure to form a single executable program.

All records passed to a PL/I procedure from the sort/merge program, and all records passed to the sort/merge program, must be in the form of character strings. Thus, if a PL/I procedure invoked via one of the sort/merge user exits receives records from the sort, it must include a character-string parameter; if it passes records to the sort, it must include a RETURN statement with a character-string expression.

A PL/I procedure invoked via a sort/merge user exit must pass a return code to the sort program to indicate what action should be taken when the PL/I procedure next relinquishes control. This is effected by invoking from within the procedure the PL/I library interface module PLIRETC as follows:

```
  CALL PLIRETC(n);
```

where n can have one of the following values to indicate the required action:

For procedures invoked via user exit E15:

8	Do not return to this procedure.
12	Include the record returned from the procedure in the sort.
16	Stop the sort and return immediately to the invoking procedure.

For procedures invoked via user exit E35:

4	Pass the next sorted record to the E35 procedure.
8	Do not return to this procedure.
12	Include the record returned from the procedure in the data set SORTOUT.
16	Stop the sort and return immediately to the invoking procedure.

DATA SETS USED BY SORT/MERGE

The execution step for a PL/I program that uses the PL/I sorting facilities requires some or all of the following job control statements, in addition to the job control statements for the PL/I program, to define the data sets used by the sort/merge program. Figure 86 on page 236 shows the file names and symbolic device names used by the sort/merge program.

Input Data Sets

If the sort/merge program is to read the records to be sorted from one or more data sets, include the following statements. (The sort/merge program can obtain records from up to nine data sets successively.)

TLBL OR DLBL STATEMENTS: For each input data set on magnetic tape with standard labels, include a TLBL statement. For each input data set on a direct-access device, include a DLBL statement. These statements must specify, in consecutive ascending sequence, the file names SORTIN1 through SORTIN9, according to the number of input data sets.

EXTENT STATEMENTS: Include an EXTENT statement for each direct-access extent occupied by the input data sets.

SORT STATEMENT: If more than on input data set is to be accessed for records to be sorted, specify the number of data sets in the FILES option of the SORT statement.

Work Data Sets

The sort/merge program requires at least three magnetic or direct-access data sets for use as intermediate storage; you can increase efficiency by specifying the direct-access data sets on separate direct-access devices. If the volume of records to be sorted demands more intermediate storage, you can specify up to nine magnetic-tape or eight direct-access data sets. Specify the number of work data sets to be used in the WORK operand of the sort/merge SORT statement.

TLBL OR DLBL STATEMENTS: For each work data set on magnetic tape with standard labels, include a TLBL statement. For each work data set on direct-access storage, include a DLBL statement.

Use of Device	**Filename**	**Symbolic Device Names When:**			
		Sort/Merge Reads Input and Writes Output	User Routine at E15 Reads Input	User Routine at E35 Writes Output	User Routines Read Input and Write Output
Output	SORTOUT	SYS001	SYS001		
Input	SORTIN1	SYS002		SYS001	
	.	.			
	.	.			
	SORTINn	SYS(n+1)		SYS(n)	
Work	SORTWK1	SYS(n+2)	SYS002	SYS(n+1)	SYS001
	.	.	.	.	.
	SORTWKm	SYS(n+m+1)	SYS(m+1)	SYS(n+m)	SYS(m)
ALTWK	SORTALT	SYS(n+m+2)	SYS(m+2)	SYS(n+n+m+1)	SYS(m+1)
CHECKPOINT	SORTCKP	SYS000	SYS000	SYS000	SYS000
n=the number of input files, as specified in the FILES parameter of the SORT statement. m=the number of work files, as specified in the WORK parameter of the SORT card.					

Figure 86. Sort/Merge File Names and Symbolic Devices

EXTENT STATEMENT: Include one extent statement for each direct-access extent used for each work data set on a direct-access device.

Output Data Sets

If the sort/merge program is to write the sorted records onto an output data set, include the following statements.

TLBL AND DLBL STATEMENTS: If the output data set is on magnetic tape with standard labels, include a TLBL statement. If the data set is on a direct-access device, include a DLBL statement. Either statement must specify the name SORTOUT for its file name.

EXTENT STATEMENT: If the output data set is on a direct-access device, include one or more EXTENT statements.

Symbolic Device Names

The symbolic device names used in these statements must be determined according to the information in Figure 86 on page 236.

INVOKING SORT/MERGE FROM PL/I

The sort/merge program is invoked from a PL/I program by one of the CALL statements listed below. The number of arguments required depends on the entry name invoked.

The arguments include sort/merge program control statements that define the processing to be carried out and describe the records to be sorted. (When the sort/merge program is invoked as an independent job step, these control statements are submitted via SYSRDR.) The control statements are described in the appropriate sort/merge publication for the version of the sort/merge program to be used. Note that the PL/I sort interface of the sort/merge program (5746-SM2) does not permit the use of the MERGE statement.

The general syntax of the CALL statement for each of the four entry points is:

```
  CALL PLISRTA(arg1,arg2,arg3,arg4[,arg7
               [,arg8]]);

  CALL PLISRTB(arg1,arg2,arg3,arg4,arg5
               [,arg7[,arg8]]);

  CALL PLISRTC(arg1,arg2,arg3,arg4,arg6
               [,arg7[,arg8]]);

  CALL PLISRTD(arg1,arg2,arg3,arg4,arg5,
               arg6[,arg7[,arg8]]);
```

The arguments are:

arg1	Sort/merge SORT statement.
arg2	Sort/merge RECORD statement.
arg3	Sort/merge OPTION statement.
arg4	Name of variable in invoking procedure that is to receive the sort return-code value.
arg5	Entry name of the PL/I procedure to be invoked from user exit E15.
arg6	Entry name of the PL/I procedure to be invoked from user exit E35.
arg7	Sort/merge INPFIL statement.
arg8	Sort/merge OUTFIL statement.

Control statements for the sort/merge program that are arguments to one of the PL/I-sort interfaces are represented in the PL/I program as character strings. Unless they are coded as null strings, these statements must be preceded and followed by a blank character.

The following notes apply to the use of the OPTION statement, the INPFIL statement, and the OUTFIL statement.

1. The OPTION Statement: The STORAGE option of the OPTION statement must be specified unless the minimum size of storage is acceptable for the sort/merge program. The minimum storage requirement of the sort program is 16K bytes. If a direct-access storage device is to be used for intermediate storage, at least 32K bytes of storage must be available. These storage requirements are in addition to the storage requirements of the PL/I program. The STORAGE option must have the format:

   ```
   STORAGE=(n|nK)
   ```

   where n is an integer. The first parameter of the OPTION statement should be PRINT=NONE, and the parameters ROUTE=LST and DUMP should not be used. If it is necessary to use PRINT=ALL, PRINT=CRITICAL, DUMP, or ROUTE=LST, then SYSLST must be closed when PLISRT is called, and the SORT user exits must not use SYSLST. However, if one of these parameters is used and for any reason PL/I error routines get control, an ABEND will occur.

2. The INPFIL Statement: The EXIT option must be specified if you are using either PLISRTB or PLISRTD with a PL/I procedure at user exit E15 to obtain records and pass them to the sort/merge program. Do not use this option if the sort/merge program is to obtain records for sorting from a SORTIN input data set even if such records are made available to a PL/I procedure invoked via user exit E15.

3. The OUTFIL Statement: The EXIT option must be specified if you are using either PLISRTC or PLISRTD to pass all the sorted records to a PL/I procedure at user exit E35 and a SORTOUT data set is not required. Do not use this option if the sort program is to write any records onto a SORTOUT data set even if such records are made available to a PL/I procedure invoked via user exit E35.

4. In those cases in which an OPTION statement is neither necessary nor desired, arg3 may be coded as a null string. The argument must always be coded even though it is not needed; it must not be omitted.

5. In those cases in which an INPFIL statement is neither necessary nor desired, arg7 may be coded as a null string. The argument must be coded if arg8 is also coded; arg7 may be omitted only if arg8 is also omitted.

6. In those cases in which an OUTFIL statement is neither necessary nor desired, arg8 may be coded either as a null string or omitted entirely.

EXAMPLES OF USING PL/I SORT

The following examples make use of a job stream using sort on disk.

SORTING RECORDS DIRECTLY FROM ONE DATA SET TO ANOTHER (PLISRTA)

The example in Figure 87 on page 239 illustrates the use of entry point PLISRTA to retrieve records from an input data set (SORTIN1), sort them, and write them directly in sorted sequence onto an output data set (SORTOUT). The example will sort records that are 80 bytes in length. The sort field commences on byte 7 and is the remaining 74 bytes of the record. The records are sorted into ascending alphameric sequence.

The PL/I program contains the following elements:

- A declaration of the variable RETURN_CODE to receive the return code from the sort/merge program.
- A CALL statement to invoke the entry point PLISRTA.
- Statements to test the return code.

The example uses the minimum of data sets: one for input, one of output, and three direct-access storage extents on a single disk storage drive.

```
// JOB FIG1402
// OPTION LINK
// EXEC PLIOPT,SIZE=100K
        /* PL/I PROGRAMMING EXAMPLE USING PLISRTA */
 EX106: PROCEDURE OPTIONS(MAIN);

              DCL RETURN_CODE FIXED BINARY(31,0);

         /* INVOKE THE SORT PROGRAM  */

         CALL PLISRTA (' SORT FIELDS=(7,74,CH,A,),WORK=3 ',
                       ' RECORD TYPE=F, LENGTH=(80) ',
                       ' OPTION PRINT=NONE,STORAGE=45000 ',
                       RETURN_CODE,
                       ' INPFIL BLKSIZE=80 ',
                       ' OUTFIL BLKSIZE=80 ');

         /* TEST RETURN CODE */

        IF RETURN_CODE=16
           THEN PUT SKIP EDIT('SORT FAILED')(A);
           ELSE IF RETURN_CODE=0
              THEN PUT SKIP EDIT('SORT COMPLETE')(A);
              ELSE PUT SKIP EDIT
                   ('INVALID SORT RETURN CODE')(A);
        /* SET RETURN CODE TO REFLECT SUCCESS OF SORT */
        CALL PLIRETC(RETURN_CODE);
        END EX106;
/*
// EXEC LNKEDT
// ASSGN SYS001,3330,VOL=DOS222,SHR
// ASSGN SYS002,3330,VOL=DOS222,SHR
// ASSGN SYS003,3330,VOL=DOS222,SHR
// ASSGN SYS004,3330,VOL=DOS222,SHR
// ASSGN SYS005,3330,VOL=DOS222,SHR
// DLBL SORTOUT,'SDATA',0,SD
// EXTENT SYS001,DOS222,1,0,3990,19
// DLBL SORTIN1,'DDATA',0,SD
// EXTENT SYS002,DOS222,1,0,4009,19
// DLBL SORTWK1,,0
// EXTENT SYS003,DOS222,1,0,4028,19
// DLBL SORTWK2,,0
// EXTENT SYS004,DOS222,1,0,4047,19
// DLBL SORTWK3,,0
// EXTENT SYS005,DOS222,1,0,4066,19
// EXEC ,SIZE=100K
/&
```

Figure 87. Using PL/I to Invoke Sort/Merge (PLISRTA)

USER EXIT E15 (PLISRTB)

The example in Figure 88 on page 241 illustrates the use of entry point PLISRTB to enable records to be supplied to the sort by a PL/I procedure.

Like that in the previous example, the main procedure invokes the sort program and test the return code when processing is complete. The presence of the EXIT option in the INPFIL statement indicates to the sort/merge program that all records to be sorted will be supplied by the procedure invoked via user exit E15 (in this case, procedure E15X).

Each time procedure E15X is invoked by the sort/merge program, it reads a record from the input stream and passes it to the sort after the appropriate return code has been passed.

USING USER EXIT E35 TO HANDLE SORTED RECORDS

The example in Figure 89 on page 242 illustrates the use of entry point PLISRTC to enable records to be supplied from the sort to the PL/I procedure. As in previous examples, the main procedure invokes the sort program and tests the return code when processing is complete. The presence of the EXIT option in the OUTFIL statement indicates to the sort/merge program that all records to be sorted are to be passed to the procedure invoked by user-exit E35 (in this case, procedure E35X). Each time procedure E35X is invoked by the sort/merge program, it receives a sorted record as a parameter, prints it, and requests the next record from the sort/merge program by passing it the appropriate return code.

PASSING RECORDS TO BE SORTED, AND RECEIVING SORTED RECORDS (PLISRTD)

The example in Figure 90 on page 243 illustrates the use of entry point PLISRTD to enable records to be supplied to the sort from a PL/I procedure and sorted records to be supplied from the sort to a PL/I procedure. As in previous examples, the main procedure invokes the sort program and tests the return code when processing is complete. The use of the E15 user exit is similar to that in Figure 88 on page 241; the use of the E35 user exit is similar to that in Figure 89.

The sequence of events is a follows:

1. The PL/I program invokes the sort/merge program.
2. The sort/merge program invokes the E15 routine for each input record until the return code is set to 8.
3. The records are sorted.
4. The sort/merge program invokes the E35 routine for each sorted record until all the sorted records have been passed or until the E35 routine requests no more records.

<u>SORTING VARIABLE-LENGTH RECORDS</u>

The PL/I-sort interface facilities can be used to sort variable-length records in the following circumstances:

- By using PLISRTA to obtain variable-length records from a SORTIN data set, and transmit variable-length records to a SORTOUT data set.
- By using PLISRTC to obtain variable-length records from a SORTIN data set and pass them as <u>adjustable</u> strings to an E35 routine.

```
// JOB FIG1403
// OPTION
// EXEC PLIOPT,SIZE=100K  /* PL/I PROGRAMMING EXAMPLE USING PLISRTB */

 EX107:  PROC OPTIONS(MAIN);
         DCL RETURN_CODE FIXED BINARY(31,0);  /* INVOKE THE SORT PROGRAM */

           CALL PLISRTB (' SORT FIELDS=(7,74,CH,A),WORK=4 ',
                    ' RECORD TYPE=F,LENGTH=(80) ',
                    ' OPTION PRINT=NONE,STORAGE=45000 ',
                    RETURN_CODE,
                    E15X,
                    ' INPFIL EXIT ',
                    ' OUTFIL BLKSIZE=80 ');
        /*  TEST RETURN CODE  */
     IF RETURN_CODE=16
        THEN PUT SKIP EDIT('SORT FAILED')(A);
        ELSE IF RETURN_CODE=0
           THEN PUT SKIP EDIT('SORT COMPLETE')(A);
           ELSE PUT SKIP EDIT('INVALID SORT RETURN CODE')(A);
      /* SET THE RETURN CODE TO REFLECT SUCCESS OF SORT */
        CALL PLIRETC(RETURN_CODE);

        E15X:   /*  THIS PROCEDURE OBTAINS RECORDS FROM THE INPUT STREAM */

           PROC RETURNS(CHAR(80));
                DCL SYSIN FILE RECORD INPUT ENVIRONMENT
                       (F RECSIZE(80)MEDIUM(SYSIPT));
                ON ENDFILE(SYSIN) BEGIN;
                  PUT SKIP(3) EDIT
                       ('END OF SORT PROGRAM INPUT'((A);
                  CALL PLIRETC(8);  /*  SIGNAL END OF SORT INPUT  */
                  GOTO ENDE15;
                  END;
                DCL INFIELD CHAR(80);
                READ FILE (SYSIN) INTO (INFIELD);
                CALL PLIRETC(12);  /*  INPUT TO SORT CONTINUES  */
                RETURN (INFIELD);
        ENDE15: END E15X;
 END EX107;
/*
// EXEC LNKEDT
// ASSGN SYS001,3330,VOL=DOS222,SHR
// ASSGN SYS002,3330,VOL=DOS222,SHR
// ASSGN SYS003,3330,VOL=DOS222,SHR
// ASSGN SYS004,3330,VOL=DOS222,SHR
// ASSGN SYS005,3330,VOL=DOS222,SHR
// DLBL SORTOUT,'SDATA',0
// EXTENT SYS001,DOS222,1,0,3990,19
// DLBL SORTWK1,,0
// EXTENT SYS002,DOS222,1,0,4009,19
// DLBL SORTWK2,,0
// EXTENT SYS003,DOS222,1,0,4028,19
// DLBL SORTWK3,,0
// EXTENT SYS004,DOS222,1,0,4047,19
// DLBL SORTWK4,,0
// EXTENT SYS005,DOS222,1,0,4066,19
// EXEC ,SIZE=100K
003329HOOKER S.W. RIVERDALE, SATCHWELL LANE, BACONSFIELD
002886BOOKER R.R. ROTORUA, MILKEDGE LAND, TOBLEY
003077ROOKER & SON, LITTLETON NURSERIES, SHOLTSPAR
059334HOOK E.H. 109 ELMTREE ROAD, GANNET PARK, NORTHAMPTON
073872HOME TAVERN, WESTLEIGH
/*
/&
```

Figure 88. Using User Exit E15 to Supply Records for Sorting (PLISRTB)

```
// JOB FIG1404
// OPTION LINK
// EXEC PLIOPT,SIZE=100K
       /* PL/I PROGRAMMING EXAMPLE USING PLISRTC */
 EX108:  PROCEDURE OPTIONS(MAIN)

         DCL RETURN_CODE FIXED BINARY(31,0);

         /* INVOKE THE SORT PROGRAM */

            CALL PLISRTC (' SORT FIELDS=(7,74,CH,A),WORK=4 ',
                   ' RECORD TYPE=F,LENGTH=(80) ',
                   ' OPTION PRINT=NONE,STORAGE=45000 ',
                   RETURN_CODE,
                   E35X,
                   ' INPFIL BLKSIZE=80 ',
                   ' OUTFIL EXIT ');

         /*  TEST RETURN CODE  */

     IF RETURN_CODE=16
        THEN PUT SKIP EDIT('SORT FAILED')(A);
        ELSE IF RETURN_CODE=0
           THEN PUT SKIP EDIT('SORT COMPLETE')(A);
           ELSE PUT SKIP EDIT('INVALID SORT RETURN CODE')(A);
      /* SET THE RETURN CODE TO REFLECT SUCCESS OF SORT */
        CALL PLIRETC(RETURN_CODE);

         E35X:   /* THIS PROCEDURE OBTAINS SORTED RECORDS */

       PROC (INREC);  /* PROCESS SORTED RECORDS  */

            DCL INREC CHAR(80);
            PUT SKIP EDIT (INREC) (A);
            CALL PLIRETC(4);  /* REQUEST NEXT RECORD FROM SORT */
            END E35X;
         END EX108;
/*
// EXEC LNKEDT
// ASSGN SYS001,3330,VOL=DOS222,SHR
// ASSGN SYS002,3330,VOL=DOS222,SHR
// ASSGN SYS003,3330,VOL=DOS222,SHR
// ASSGN SYS004,3330,VOL=DOS222,SHR
// ASSGN SYS005,3330,VOL=DOS222,SHR
// DLBL SORTIN1,'DDATA',0
// EXTENT SYS001,DOS222,1,0,3990,19
// DLBL SORTWK1,,0
// EXTENT SYS002,DOS222,1,0,4009,19
// DLBL SORTWK2,,0
// EXTENT SYS003,DOS222,1,0,4028,19
// DLBL SORTWK3,,0
// EXTENT SYS004,DOS222,1,0,4047,19
// DLBL SORTWK4,,0
// EXTENT SYS005,DOS222,1,0,4066,19
// EXEC ,SIZE=100K
/&
```

Figure 89. Using User Exit E35 to Handle Sorted Records (PLISRTC)

```
// JOB FIG1405
// OPTION LINK
// EXEC PLIOPT,SIZE=100K
 EX109:  PROC OPTIONS(MAIN);  /* PL/I PROGRAMMING EXAMPLE USING PLISRTD */
         DCL RETURN_CODE FIXED BINARY(31,0);
         /* INVOKE THE SORT PROGRAM */
            CALL PLISRTD (' SORT FIELDS=(7,74,CH,A),WORK=4 ',
                   ' RECORD TYPE=F,LENGTH=(80) ',
                   ' OPTION PRINT=NONE,STORAGE=45000 ',
                   RETURN_CODE,
                   E15X,
                   E35X,
                   ' INPFIL EXIT ',
                   ' OUTFIL EXIT ');
         /*  TEST RETURN CODE  */
     IF RETURN_CODE=16 THEN PUT SKIP EDIT ('SORT FAILED')(A);
     ELSE IF RETURN_CODE=0
          THEN PUT SKIP EDIT('SORT COMPLETE')(A);
          ELSE PUT SKIP EDIT ('INVALID SORT RETURN CODE')(A);
      /* SET RETURN CODE TO REFLECT SUCCESS OF SORT */
    CALL PLIRETC(RETURN_CODE);
         E15X:   /* THIS PROCEDURE OBTAINS RECORDS FROM THE INPUT STREAM */
           PROC RETURNS(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 SORT INPUT  */
                GOTO ENDE15;
                END;
               DCL INFIELD CHAR(80);
               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: /* THIS PROCEDURE OBTAINS SORTED RECORDS */
       PROC (INREC);
           /* PROCESS SORTED RECORDS  */
            DCL INREC CHAR(80);
            PUT SKIP EDIT (INREC) (A);
            CALL PLIRETC(4); /* REQUEST NEXT RECORD FROM SORT */
            END E35X;
         END EX109;
/*
// EXEC LNKEDT
// ASSGN SYS001,3330,VOL=DOS222,SHR
// ASSGN SYS002,3330,VOL=DOS222,SHR
// ASSGN SYS003,3330,VOL=DOS222,SHR
// ASSGN SYS004,3330,VOL=DOS222,SHR
// DLBL SORTWK1,,0
// EXTENT SYS001,DOS222,1,0,3990,19
// DLBL SORTWK2,,0
// EXTENT SYS002,DOS222,1,0,4009,19
// DLBL SORTWK3,,0
// EXTENT SYS003,DOS222,1,0.4028,19
// DLBL SORTWK4,,0
// EXTENT SYS004,DOS222,1,0,4047,19
// EXEC ,SIZE=100K
003329HOOKER S.W. RIVERDALE, SATCHWELL LANE, BACONSFIELD
002996BOOKER S.W. ROTORUA, MILKEDGE LANE, TOBLEY
003077ROOKER & SON, LITTLETON NURSERIES, SHOLTSPAR
059334HOOK E.H. 109 ELMTREE ROAD, GANNET PARK, NORTHAMPTON
073872HOME TAVERN, WESTLEIGH
000931FOREST,IVER,BUCKS
/*
/&
```

Figure 90. Using User Exits E15 and E35 (PLISRTD)

- By using PLISRTB to obtain varying-length strings from an E15 routine with the option RETURNS(CHAR(n)VARYING), convert them to variable-length records, sort them, and transmit them to a SORTOUT data set.

- By using PLISRTD to obtain varying-length strings from an E15 routine with the option RETURNS(CHAR(n)VARYING), convert then to a variable-length records, sort them, and pass them reconverted as adjustable strings to an E35 routine.

An E15 routine can also be used as follows:

- Fixed-length strings can be returned from the PL/I program to be sorted in a sorting operation for which variable-length records are specified.

- Varying-length strings can be returned from the PL/I program to be sorted in a sorting operation for which fixed-length records are specified.

In the first case, the PL/I interface converts the fixed-length strings to variable-length records of equal length. In the second case, the varying-length strings are padded with blanks to the maximum length and converted to fixed-length records.

(Note that an E35 routine cannot be used in a similar manner. If the sorted records passed to it are fixed-length, a fixed-length string must be used; if the sorted records are variable-length, an adjustable string must be used.)

Variable-length records can be sorted, provided that the position and length of fields that are used as sort keys are identical for every record.

An example of a PL/I program to invoke sort/merge to sort varying-length strings as variable-length records is given in Figure 91 on page 245. The example uses PLISRTB to pass varying-length strings to the sort program. The sort program sorts them and writes them onto the data set VRECS. The records have a maximum length of 84 bytes including a 4-byte length field added to each varying-length string before it is passed to the sort program. Consequently the sort field, which commences on the seventh position in the varying-length string, is specified in the SORT FIELDS statement as starting in the eleventh position in the variable-length record. The sort field is the following 14 characters, and lies within the minimum record length of 24.

The program lists each record in the input stream and converts it into a varying-length string. It also prints the length of the data portion of each string that is processed.

Variable-length records passed to an E35 exit routine must be declared as parameters that are adjustable character strings. The use of the VARYING attribute for such parameters is not permitted. For example, suppose the records are variable length and unblocked, with the following characteristics:

Maximum length for both input and output:	80 bytes
Minimum length for both input and output:	20 bytes
Most frequent record length:	40 bytes
Maximum block size:	84 bytes

An example of a PL/I program to invoke sort/merge to sort such records is given in Figure 92 on page 246. In the example, PLISRTC obtains variable-length records from the data set VRECS (created in the example in Figure 91), sorts them, and passes them as adjustable strings to the E35 routine which prints them.

```
// JOB FIGI306
// OPTION LINK
// EXEC PLIOPT,SIZE=100K
 /*PL/I EXAMPLE USING PLISRTB TO SORT VARYING-LENGTH RECORDS*/
    EX1406:  PROC OPTIONS(MAIN);
             DCL RETURN_CODE FIXED BIN(31,0);
             CALL PLISRTB (' SORT FIELDS=(11,14,CH,A),WORK=4 ',
                          ' RECORD TYPE=V,LENGTH=(84,,,24,44) ',
                          ' OPTION PRINT=NONE,STORAGE=45000 ',
                          RETURN_CODE,
                          E15X,
                          ' INPFIL EXIT ',
                          ' OUTFIL BLKSIZE=88 ');
             IF RETURN_CODE=0
                THEN PUT SKIP EDIT ('SORT COMPLETE')(A);
                ELSE IF RETURN_CODE=16
                     THEN PUT SKIP EDIT('SORT FAILED')(A);
                     ELSE PUT SKIP EDIT
                          ('INVALID RETURN CODE')(A);
           /* SET RETURN CODE TO REFLECT SUCCESS OF SORT */
      CALL PLIRETC(RETURN_CODE);

    E15X:    PROC RETURNS (CHAR(80) VARYING);
             DCL STRING CHAR(80) VAR;
             ON ENDFILE(SYSIN) BEGIN;
                PUT SKIP EDIT ('END OF INPUT')(A);
                CALL PLIRETC(8);
                GOTO ENDE15;
                END;
             GET EDIT(STRING)(A(80));
             I=INDEX(STRING||'  ','  ')-1;  /* RESET THE LENGTH OF THE */
             STRING = SUBSTR(STRING,1,I);   /* STRING FROM 80 TO LENGTH */
                                            /* OF THE TEXT IN EACH INPUT*/
                                            /* RECORD                   */
             PUT SKIP EDIT(I,STRING) (F(2),X(3),A);
             CALL PLIRETC(12);
             RETURN(STRING);
    ENDE15:  END E15X;
             END EX1406;
/*
// EXEC LNKEDT
// ASSGN SYS001,3330,VOL=DOS222,SHR
// ASSGN SYS002,3330,VOL=DOS222,SHR
// ASSGN SYS003,3330,VOL=DOS222,SHR
// ASSGN SYS004,3330,VOL=DOS222,SHR
// ASSGN SYS005,3330,VOL=DOS222,SHR
// DLBL SORTOUT,'VRECS',0
// EXTENT SYS001,DOS222,1,0,3990,19
// DLBL SORTWK1,,0
// EXTENT SYS002,DOS222,1,0,4009,19
// DLBL SORTWK2,,0
// EXTENT SYS003,DOS222,1,0,4028,19
// DLBL SORTWK3,,0
// EXTENT SYS004,DOS222,1,0,4047,19
// DLBL SORTWK4,,0
// EXTENT SYS005,DOS222,1,0,4066,19
// EXEC ,SIZE=100K
003329HOOKER S.W. RIVERDALE, SATCHWELL LANE, BACONSFIELD
002886BAKER R.R. ROTORUA, MILKEDGE LANE, TOBLEY
003077ROKER & SON, LITTLETON NURSERIES, SHOLTSPAR
059334HAWICK E.H. 109 ELMTREE RD., GANNET PARK, NORTHAMPTON
073872HAWICK ARMS, WESTLEIGH
093131BIRCHWOOD,FOREST & CO. IVER, BUCKS
/*
/&
```

Figure 91. Using PLISRTB to Sort Varying-Length Strings

The records have a maximum length of 84 bytes and a maximum block size of 88 bytes. Each record has a 4-byte length field which remains with it throughout the sorting operation. Consequently the sort field, which commences on the first byte of data part of each record, is specified in the SORT FIELDS statement as starting in the fifth position on the record. The sort field is six characters in length and is to be sorted in ascending sequence.

The sorted records are passed to the PL/I E35 routine, their length fields having been removed by the PL/I-sort interface routine.

```
//JOB FIG1407
// OPTION LINK,DUMP
// EXEC PLIOPT,SIZE=100K
        /* PL/I PROGRAMMING EXAMPLE USING PLISRTC TO SORT
           VARYING-LENGTH RECORDS
  EX1010: PROC OPTIONS(MAIN);
           DCL RETURN_CODE FIXED BINARY(31,0);
           CALL PLISRTC(' SORT FIELDS=5,6,CH,A),WORK=4 ',
                      ' RECORD TYPE=V,LENGTH=(84,,,24,44) ',
                      ' OPTION PRINT=NONE,STORAGE=45000 ',
                      RETURN_CODE,
                      E35X,
                      ' INPFIL BLKSIZE=88 ',
                      ' OUTFIL EXIT ');
        /* SET RETURN CODE TO REFLECT SUCCESS OF SORT */
     CALL PLIRETC(RETURN_CODE);

           IF RETURN_CODE=0
               THEN PUT SKIP EDIT('SORT COMPLETE')(A);
               ELSE IF RETURN_CODE=16
                    THEN PUT SKIP EDIT('SORT FAILED')(A);
                    ELSE PUT SKIP EDIT
                          ('IVALID SORT RETURN CODE')(A);
      E35X:PROC(STRING);
           DCL STRING CHAR(*);
           PUT SKIP EDIT (STRING)(A);
           CALL PLIRETC(4);
           END E35X;
      END  EX1010;
/*
// EXEC LNKEDT
// ASSGN SYS001,3330,VOL=DOS222,SHR
// ASSGN SYS002,3330,VOL=DOS222,SHR
// ASSGN SYS003,3330,VOL=DOS222,SHR
// ASSGN SYS004,3330,VOL=DOS222,SHR
// ASSGN SYS005,3330,VOL=DOS222,SHR
// DLBL SORTIN1,'VRECS'
// EXTENT SYS001,DOS222,1,0,3990,19
// DLBL SORTWK1,,0
// EXTENT SYS002,DOS222,1,0,4009,19
// DLBL SORTWK2,,0
// EXTENT SYS003,DOS222,1,0,4028,19
// DLBL SORTWK3,,0
// EXTENT SYS004,DOS222,1,0,4047,19
// DLBL SORTWK4,,0
// EXTENT SYS005,DOS222,1,0,4066,19
// EXEC ,SIZE=100K
/&
```

Figure 92. Sorting Variable-Length Records (PLISRTC)

CHAPTER 15. COMMUNICATION WITH COBOL, FORTRAN, AND RPG

PL/I permits communication, at execution time, between programs compiled by the PL/I Optimizing Compiler and programs compiled by one of the following compilers, and executed using the corresponding library.

Program	Program No.
DOS FORTRAN IV Compiler	360N-FO-479
DOS FULL ANS COBOL (Version 3) Compiler and Library	5736-CB2
(Library only)	5736-LM2
DOS/VS COBOL Compiler and Library	5746-CB1
(Library only)	5746-LM4
DOS RPG II Compiler	5736-RG1
DOS/VS RPG II Compiler	5746-RG1

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" on page 104 in Chapter 7.

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.

A COMMON block in FORTRAN 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.

A program compiled by the DOS RPG II compiler can invoke a PL/I routine from its calculation section by means of an EXIT operation. Parameters cannot be passed, nor can values be returned, but data can be communicated by means of static storage. (See "Matching RPG Arguments/Parameters" on page 256).

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. Existing RPG II programs designed to invoke assembler routines can invoke PL/I routines without modification.

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.

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 for COBOL and FORTRAN, and to achieve correspondence between PL/I declarations and RLABL and ULABL operations for RPG II.

For COBOL and FORTRAN, the interlanguage facilities automatically 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. Because 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 "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 below, in Figure 93 on page 254 and Figure 95 on page 256, respectively.

PROGRAM-CONTROL DATA: Arguments of any program-control data type can be passed to an invoked COBOL or FORTRAN routine. However, only 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 restrictions given above. In addition, each member can have its own alignment restriction or all members can have the same alignment restriction. 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.

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 mapping of 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 argument 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) a 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.

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.

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" later in this chapter).

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 or 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, the leftmost 8 only are taken.

Examples

```
1. DCL COBOL ENTRY (CHAR(5))
        OPTIONS(COBOL INTER),

       COBOLB ENTRY (1, 2 FIXED, 2 FLOAT)
        OPTIONS(COBOL NOMAPIN),
       COBOLBXX 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(1 UNALIGNED, 2 FIXED,
        2 FLOAT)
          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 COSUB 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. Because 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 argument 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 may 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

The entry points in a PL/I procedure that are to be invoked from COBOL, FORTRAN, or RPG 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, and, for RPG, that identify the names of common data areas.

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.

RPG
This specifies that the entry point can only be invoked by an RPG 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.

RLABL
This specifies that the variables named in the option are STATIC EXTERNAL variables used for communication with RPG and that they also appear in RPG RLABL operations.

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, FORTRAN, or RPG 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 or an RPG routine.

- The parameter or parameters to which the NOMAP, MAPIN, or MAPOUT options apply can be specified in an argument list.

- An entry point that is to be invoked by an RPG routine cannot have parameters.

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 ...,
            1 Y,  2 ...;
   ```

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 93. These are the PL/I data types that should appear in PL/I parameter descriptors associated with COBOL arguments or parameters, respectively.

COBOL				PL/I			
		ALIGNMENT				ALIGNMENT	
DATA TYPE	LENGTH (BYTES)	SYNCH. (ALIGNED)	UNSYNCH. (un-ALIGNED)	DATA TYPE	LENGTH (BYTES)	ALIGNED	UN-ALIGNED
COMPUTATIONAL[1] dec. length: 1-4	2	Halfword	Byte	FIXED BINARY(15,0) (halfword integer)	2	Half-word	Byte
5-9	4	Fullword	Byte	FIXED BINARY(31,0) (fullword integer)	4	Full-word	Byte
10-18	8	Fullword	Byte	No equiva-lent	—	—	—
COMPUTA-TIONAL-1	4	Fullword	Byte	FLOAT DEC(6) (short float)	4	Full-word	Byte
COMPUTA-TIONAL-2	8	Double-word	Byte	FLOAT DEC(16) (long float)	8	Double-word	Byte
COMPUTA-TIONAL-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.

Figure 93. COBOL-PL/I Data Equivalents

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 94 on page 255 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.

```
COBOL                                          PL/I

01  A SYNCHRONIZED.                            1  A ALIGNED,
    02  B OCCURS 3 TIMES.                         2  B(3),
        03  C OCCURS 4 TIMES.                        3  C(4),
            04  D OCCURS 5 TIMES USAGE COMP-3           4  D(5) FIXED
                                                           DECIMAL(7,3),
                PIC S9999V999.
    02  E USAGE DISPLAY.                          2  E,
        03  F PIC X(8).                              3  F CHAR(8),
        03  G PIC 9(8).                              3  G PIC '(8)9',
    02  DUMMY OCCURS 6 TIMES.                     2  H(6,7) FIXED BINARY
                                                           (15,0);
        03  H OCCURS 7 TIMES USAGE COMP
                PIC S9999.
```

Figure 94. Declaration of a Data Aggregate in COBOL and PL/I

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 automatically 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 95 on page 256. 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.

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

FORTRAN			PL/I			
					ALIGNMENT	
DATA TYPE	LENGTH (BYTES)	ALIGNMENT[1]	DATA TYPE	LENGTH (BYTES)	ALIGNED	UN-ALIGNED
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	Double-word	REAL FLOAT DEC(16) (real long float)	8	Double-	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
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 restriction.

[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 95. FORTRAN-PL/I Data Equivalents

message is produced and the array is not automatically remapped. Similarly, only these lengths should be used for PL/I variables having storage in common with FORTRAN variables.

MATCHING RPG ARGUMENTS/PARAMETERS

A PL/I procedure or entry point that is to be invoked from RPG II must have OPTIONS(RPG) specified on the PROCEDURE or ENTRY statement. The PROCEDURE or ENTRY statement must be in the outermost block of the PL/I program, and no other options of the OPTIONS option, except RLABL, may appear.

The PL/I routine can be invoked from the Calculation Section of the RPG program by means of the EXIT operation. Control returns to the RPG program when the END statement of the PL/I routine is reached, or when a RETURN statement is executed in the outermost block.

It is not possible for a PL/I routine to invoke an RPG program.

The PL/I environment is established on the first invocation from the RPG program. Thereafter it remains established until the RPG program terminates.

Data can be communicated between RPG II and PL/I by means of STATIC EXTERNAL storage or by means of files. The shared data may be an RPG indicator, an RPG field, or an RPG array or table.

Data that appears in an RLABL or ULABL operation of RPG must be explicitly declared in PL/I as unqualified, level-1, STATIC EXTERNAL variables. Variables that appear in an RLABL operation must also appear in the RLABL option of the OPTIONS option of the PL/I PROCEDURE or ENTRY statement; these variables must not be declared with the INITIAL attribute.

An RPG indicator is a single byte with a 2-character identification. To reference an indicator in PL/I, the PL/I variable must be named INxx, where xx is the RPG 2-character identification, and be declared as CHARACTER(1). Indicators known to PL/I must appear in an RLABL operation in RPG, and so must also appear in the RLABL option in PL/I.

An indicator is off when its bit value is '00000000'B, and on when its bit value is '11110000'B. Because indicators must be declared as CHARACTER(1), these values are most conveniently referred to as "off" = LOW(1) and "on" = '0'.

Fields in RPG may be declared as CHARACTER or FIXED DECIMAL, according to the format of the field. The PL/I variable must have the same name as the RPG field, and the field must appear in an RLABL or ULABL operation in RPG.

An RPG table or array may be referenced in PL/I if the name of the table or array appears in an RLABL operation and the same name is declared in PL/I and appears in the RLABL option.

You must be aware of the structure of an RPG II linkage field for a table or array, and describe the corresponding object in PL/I. An example of a PL/I declaration corresponding to a table linkage field is shown in Figure 96.

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 will never prevent data being passed, nor will it attempt to correct errors; although it produces messages to indicate likely sources of error, it will always allow you to attempt to pass any type of data you specify.

```
PLISUB: PROCEDURE OPTIONS(RPG,RLABL(TABLET));

  DECLARE 1 TABLET STATIC EXTERNAL,/*RESOLVED TO RPG LINKAGE FIELD*/
            2 ELEMENT_LENGTH FIXED BINARY(15,0),
            2 NUMBER_OF_ENTRIES FIXED BINARY(15,0),
            2 START_OF_TABLE POINTER,
            2 END_OF_TABLE POINTER,
            2 LAST_LOOKUP POINTER;

  DECLARE TABLET_ELEMENT CHAR(4) BASED;

  DECLARE NEWVAL CHAR(4) STATIC EXTERNAL; /*KNOWN TO RPG*/

  DECLARE TRANS CHAR(256); /*TRANSLATION STRING KNOWN ONLY TO PL/I*/

  NEWVAL = TRANSLATE(LAST_LOOKUP->TABLET_ELEMENT,TRANS);

   (ETC.)

END PLISUB;
```

Figure 96. Example of PL/I Procedure To Be Invoked from RPG II

Figure 97 on page 259 shows the return codes generated by various types of PL/I data.

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 2 declarations of a STATIC EXTERNAL variable in different external PL/I procedures. The number of variables using common storage is not limited to 2. 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.

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 95 on page 256 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 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 98 on page 261) 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.

PL/I ATTRIBUTE	COBOL		FORTRAN	
	ARGUMENT	PARAMETER	ARGUMENT	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 97 (Part 1 of 2). Return Codes Produced by PL/I Data Types

Notes:

1. Creation of a dummy argument is suppressed: 0008
2. BIT(8) or BIT(32): 0000
   Any other length: 0008
   In the 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 FIXED BINARY (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 a¬=0: 0004
8. If item is element of a structure or is a
   minor structure: 0000
   All other cases: 0008
9. If argument is an aggregate and creation of temporary
   suppressed by NOMAP, or if argument is scalar: 0012
   If argument is aggregate and no NOMAP: 0000

Figure 97 (Part 2 of 2). Return Codes Produced by PL/I Data Types

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 263).

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.

Because 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. 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.

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.

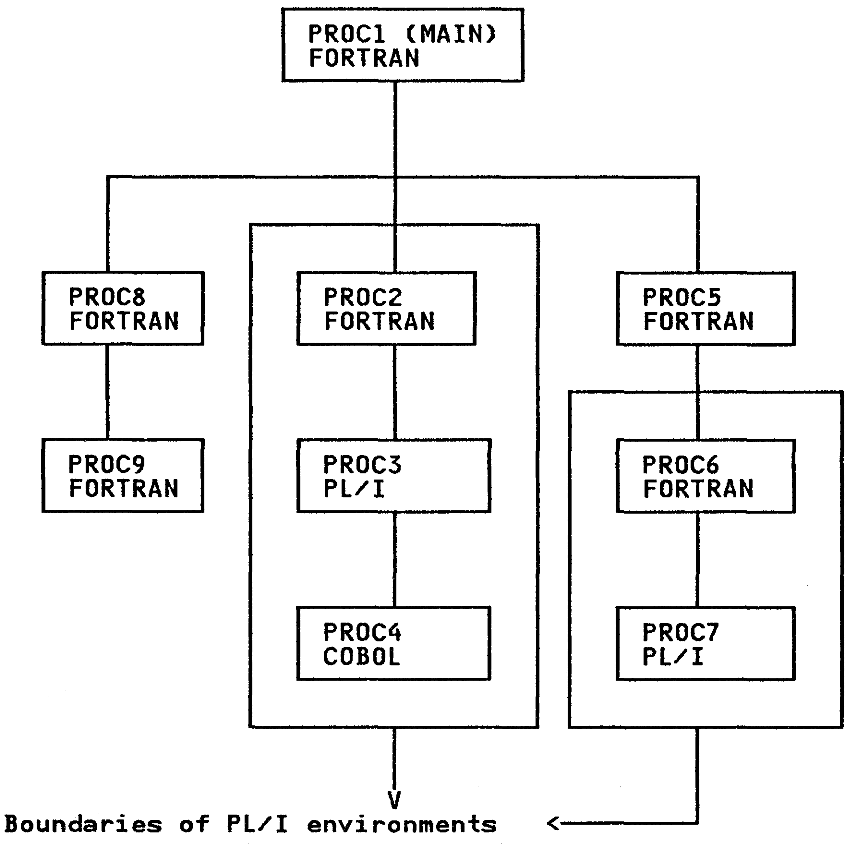


Figure 98. Extent of PL/I Environment

RESTRICTIONS ON INPUT/OUTPUT BY FORTRAN ROUTINES

Apart from the use of SYSLST, any input/output control information generated by a FORTRAN routine that has been invoked by a PL/I procedure will overwrite PL/I environment information.

Conversely, a FORTRAN routine that performs input/output and invokes a PL/I procedure will have any input/output control information overwritten by the establishment of the PL/I environment. However, an implementation-defined subroutine, PLISA, is provided to overcome this problem.

The PLISA subroutine reserves storage for the PL/I environment. Two arguments are supplied to the PLISA subroutine. The first argument is the name of the variable in which the PL/I environment is to be maintained. The second argument is an integer constant specifying the number of bytes of storage. The following sample statements could be given in a FORTRAN routine.

Note that the PLISA subroutine must be called before the PL/I procedure is called. For example:

```
  REAL*8 PLIENV(250)
     .
     .
  CALL PLISA(PLIENV,2000)
     .
  CALL PLIPROC(X)
```

Notes:

1. The FORTRAN data type REAL*8 is irrelevant in this example; any data type could be specified, provided that the array dimension is modified to maintain the same number of bytes.

2. The number of bytes given in this example should be regarded as a minimum amount of storage required by a PL/I environment. There is no simple method of determining the maximum amount of environment information required during the execution of a PL/I procedure. The safest course of action is to make the FORTRAN array variable as large as practicable.

When a PL/I procedure that has been invoked from a FORTRAN routine uses the SYSPRINT file, the last item will not be transmitted unless the file is closed before return to the FORTRAN routine. Alternatively, a PUT SKIP statement could be executed to cause transmission of the last item.

Similarly, when a PL/I procedure invokes a FORTRAN routine, the last item will not be transmitted until after return from the FORTRAN routine. Therefore, if the FORTRAN routine uses SYSLST, the last item of SYSPRINT output will be printed after the SYSLST output. To avoid this, either the SYSPRINT file should be closed or a PUT SKIP statement should be executed before the FORTRAN routine is invoked.

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 250 in this chapter.) 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. 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.

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 will be 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 the output data to be lost. Consider the example in Figure 98 on page 261; 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 descendant 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 descendant 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 with no files being closed.

When a COBOL main routine within a PL/I environment returns control to the operating system, the environment ends normally.

CHAPTER 16. USING PL/I ON CICS

PL/I can be used in conjunction with CICS facilities to write application programs for CICS/DOS/VSE. When this is done, CICS provides facilities to the PL/I program that would normally be provided directly by the Disk Operating System. These facilities include most data management facilities and all job and task management facilities.

This chapter describes the PL/I-supplied PL/I-CICS/VS interface, and the restrictions and features that apply to PL/I programs:

- Compiled on Version 1, Release 6 of the DOS PL/I Optimizing Compiler,
- Link-edited with Version 1, Release 6 of the DOS PL/I Resident Library, and
- Executed under an appropriate release of CICS/DOS/VS (for instance, Version 1, Release 4) with the CICS/VS-related modules supplied by the DOS PL/I Transient Library (Version 1, Release 6).

The information presented here and in Appendix D supplements information contained in the CICS/VS application programmer's reference manuals. Those manuals contain information on how to code programs for CICS/VS. Familiarity with them, as well as with PL/I, is assumed in the text that follows.

PL/I restrictions in the CICS/VS environment when the PL/I-supplied PL/I-CICS/VS interface is in use are enumerated in Figure 99 on page 266. PL/I restrictions imposed by the CICS-supplied PL/I-CICS interface, but relaxed by the PL/I-supplied interface, are shown in Figure 100 on page 268.

In addition to the restrictions shown in Figure 99 on page 266, considerations that apply to the following topics are discussed in this chapter:

- PL/I storage
  - Lifetime of storage acquired from CICS/VS
  - Storage classes
  - CONTROLLED storage
- 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
- Using the CICS facilities

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.

Interlanguage Communication

No communication with FORTRAN or COBOL using PL/I's interlanguage facilities.

Limited communications using OPTIONS (ASSEMBLER). See text.

Execution-Time Options

Execution-time options can only be specified in the PLIXOPT string.

Specifying the SPIE option has no effect.

Builtin Subroutines

PLISRT, PLICKPT, and PLICANC cannot be used.

PLIDUMP has certain restrictions and additional functions. See text.

PLIRETC and PLIRETV can be used to communicate between user-written programs link-edited together, but not to communicate with CICS.

Debugging Facilities

FLOW, COUNT, REPORT, 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 text.

Figure 99. PL/I Restrictions when Used with CICS

CICS-SUPPLIED INTERFACE

In early versions of CICS/DOS Entry, CICS/DOS Standard, and CICS/DOS/VS, 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 enumerate PL/I-CICS/VS restrictions, which are associated only with the PL/I-CICS interface that is supplied by CICS/DOS Entry, CICS/DOS Standard, and CICS/DOS/VS to PL/I-CICS users.

PL/I-SUPPLIED INTERFACE

In the current release of the DOS PL/I Optimizing Compiler and Libraries, PL/I supplies an interface between PL/I programs and the current release of CICS/VS. This interface, as with its CICS-supplied predecessor, consists of PL/I library modules (primarily from the PL/I Transient Library) modified for use in the CICS/VS environment. It makes substantially more of the PL/I language usable in a CICS/VS transaction program. 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.

WAYS OF WRITING CICS/VS TRANSACTIONS IN PL/I

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/VS facilities, but appear 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/VS must define application program interface protocols. These protocols occur in two forms:

- Macro-level interface
- Command-level interface

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
- 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/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). This CICS coding is supported by both CICS- and PL/I-supplied PL/I-CICS/VS interfaces, although the PL/I-supplied interface provides additional function and relaxes some restrictions on PL/I language usage. See Figure 100.

Restrictions Using CICS-Supplied Interface	**Status Using PL/I-Supplied Interface**
Statements not usable: READ, WRITE, GET, PUT, OPEN, CLOSE, DISPLAY, DELAY, REWRITE, LOCATE, DELETE, UNLOCK, STOP, HALT, EXIT, FETCH, RELEASE, WAIT.	Same, except that PUT, OPEN, and CLOSE may be issued for SYSPRINT. (See text.)
PL/I Sort/Merge not usable.	Same.
PL/I error handling not permitted. Any branch to the PL/I error handler causes CICS transaction abend, so no PL/I condition can be raised. ON, SIGNAL, and REVERT statements are not supported.	Full PL/I error handling (ON, SIGNAL, REVERT) may be used for any PL/I condition (including program checks and CICS transaction abends) that can arise under CICS/VS. See text.
PL/I options. Compiler options FLOW, COUNT, GOSTMT, and GONUMBER are not supported. No PL/I execution options are supported.	REPORT, FLOW, GONUMBER, and GOSTMT may all be used in programs using the command-level interface. They are restricted somewhat under the macro-level interface. See text.
External calls not usable.	External calls to other PL/I routines or to assembler language routines declared with OPTIONS (ASSEMBLER) may be made without restriction. Called subroutines may invoke CICS services, provided the appropriate CICS control blocks were passed to them by their callers.
Floating point arithmetic usable but: – Floating point registers not saved or restored. – Floating point registers not printed in a dump. – Floating point interrupts will cause a transaction abend. – Floating point overflow or underflow interrupts may terminate CICS.	Floating point arithmetic usable without restriction. – Floating point registers are saved and restored by the PL/I library in those few places where it is necessary. – Floating point registers are printed by PLIDUMP. – Floating point overflow and underflow may be handled in OVERFLOW and UNDERFLOW on-units. The program mask is set for PL/I and CICS/VS, respectively, as appropriate.
Names for variables used in CICS/VS macros cannot exceed 8 characters.	Same.
Multiple PL/I programs cannot be link-edited together.	Restriction removed. External calls can be used freely.
Object program size cannot exceed 256K-bytes.	Same.
Static storage not alterable if re-entrancy is to be maintained.	Same.

Figure 100 (Part 1 of 2). CICS-Supplied Interface Restrictions and PL/I-Supplied Interface Status

Restrictions Using CICS-Supplied Interface	**Status Using PL/I-Supplied Interface**
CONTROLLED variables not usable. May cause invalid CICS FREEMAIN.	Not usable. Causes non-reentrant code.
STATIC EXTERNAL variables must have the INITIAL attribute because CICS/VS cannot handle common CSECTs.	Same.

Figure 100 (Part 2 of 2). CICS-Supplied Interface Restrictions and PL/I-Supplied Interface Status

COMMAND-LEVEL INTERFACE

CICS/VS Version 1, Release 3 provided a new set of programming protocols for CICS/VS programming. This interface is invoked by coding PL/I statements of the form:

```
EXEC CICS verb parameter-list
```

in the application program, and executing a CICS/VS utility program called the CICS Translator. The Translator supplies a control block (DFHEIB) for receipt of information from CICS/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; in most cases, you need neither address nor 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/VS transaction programs in PL/I. The command-level interface is described in detail in CICS/VS Application Programmer's Reference Manual, (Command Level). Use of this interface requires the PL/I-supplied PL/I-CICS/VS interface.

Using the PL/I-supplied interface thus permits either macro-level or command-level programs, or mixtures of the two, to be written in PL/I for CICS/VS. It is strongly recommended that only command-level coding be used for new CICS/VS-PL/I programming.

Although the command-level coding protocols permit extensive validation of EXEC CICS commands, neither PL/I nor CICS/VS has any real way, under either set of CICS coding protocols, to diagnose use of the PL/I features listed as restrictions in Figure 99 on page 266. For example, the compiler would regard syntactically valid PL/I statements, such as READ, WRITE, or REWRITE, or calls to PLICKPT or PLISRTC, as 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/VS, including termination of CICS/VS itself, unpredictable transaction abends, system waits, and so on. Avoidance of restricted PL/I facilities in a CICS/VS environment is your responsibility.

With the issues concerning macro-level versus command-level coding, and the CICS- versus PL/I-supplied PL/I-CICS/VS interfaces addressed by the above text and Figure 99 and Figure 100, the remainder of this chapter is devoted entirely to the PL/I-supplied PL/I-CICS/VS interface.

PL/I STORAGE

LIFETIME OF STORAGE ACQUIRED FROM CICS/VS

When storage is acquired from CICS/VS via a CICS/VS 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/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, CICS frees the program's PL/I storage, 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 transaction work area (TWA) or communication area (COMMAREA) is available to hold a PL/I pointer to be communicated between the two. The TWA, because it is a part of the CICS task-related control block structure, remains available to both programs. CICS/VS 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 unpredictable.

If A acquired the storage by issuing a PL/I ALLOCATE statement for a PL/I BASED variable, 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/VS 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, B would have to acquire the storage by CICS GETMAIN.

STORAGE CLASSES

Because changing STATIC storage violates the integrity of reentrant procedures, the CICS user should avoid writing into STATIC storage. 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 allows re-entry and should suffice for most purposes, you can allocate and free storage via ALLOCATE and FREE statements. BASED variables can be allocated and freed in this way.

CONTROLLED storage may be used in OS CICS/VS-PL/I transactions; however, the addressing mechanism used for CONTROLLED storage on

DOS PL/I is nonreentrant, making it impractical to use CONTROLLED storage in DOS PL/I-CICS transactions. (One could put CICS ENQ/DEQ around the allocation, use, and freeing of CONTROLLED variables, and make them work on DOS.) CONTROLLED storage should thus be avoided in transaction programs that are expected to be used under both DOS and OS systems.

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.

SYSPRINT

SYSPRINT can be used for any type of stream output. It is also used for error messages generated by the program and REPORT 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 may 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 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 101.

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 hexadecimal '00'
ASA is the American National Standard carriage control character

Figure 101. Format of Records Sent to SYSPRINT

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. If a debugging system that executes one transaction at a time is used, it will not occur. In a system executing many transactions, it will. If it does occur, you must use an application program to sort the outputs of the various programs using the terminal and transaction identifiers as keys.

DECLARATION OF SYSPRINT

SYSPRINT need not be declared in the 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 may 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, 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 may not be transmitted.

The LINENO and COUNT built-in functions of PL/I stream I/O may be used against SYSPRINT. In the CICS/DOS/VS environment, however, they will return zero. Thus, you should avoid use of these built-in functions 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/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), 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. The PL/I library modules that communicate with CICS use the assembler language version of the CICS macro interface, and thus use exactly the same CICS control blocks as the user's macro-level PL/I program. If assigning to a variable in one of those control blocks raises CHECK for that variable, then the PL/I SYSPRINT transmitter, in attempting to output the CHECK information, may store into the same variable in the same control block, destroying the value set by the user program. Because 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. No such problem exists with programs coded with command-level coding.

EXECUTION-TIME OPTIONS

Under CICS, execution-time options can only be specified in a character string named PLIXOPT. For example:

```
  DCL PLIXOPT CHAR(20) VAR STATIC EXTERNAL
    INIT('ISASIZE(3000) NOSTAE');
```

The following options may be used. IBM recommended defaults are underlined.

COUNT | <u>NOCOUNT</u>
FLOW | <u>NOFLOW</u>
ISASIZE
REPORT | <u>NOREPORT</u>
<u>STAE</u> | NOSTAE

COUNT, FLOW, and REPORT depend on PL/I termination being properly performed for their correct execution. See the discussion on PL/I program termination in "Error Handling" on page 274. Using CICS/VS macro instructions may cause FLOW and REPORT, as well as the compile-time option GOSTMT, to give erroneous results.

The default options for COUNT and FLOW are taken from the options specified at compile time.

The STAE option specifies that PL/I error handling will be used for hardware-detected interrupts and CICS abends.

ISASIZE specifies the initial size of the storage obtained for PL/I use. Storage obtained will be retained by PL/I throughout the execution of the program. Further storage can be obtained if necessary, but fastest execution is achieved if all storage is obtained in the ISA.

If too small a value is specified in ISASIZE, the minimum acceptable size is acquired. If the ISASIZE option is not specified, an attempt will be made to allocate an ISA sufficiently large to include both the standard control blocks and the DSA for the main procedure. (Such an attempt may fail if the FLOW option is used.) Thus, the minimum storage will be acquired. Any other storage obtained for static and automatic variables will have to be requested from CICS. This may result in slow execution. To determine the optimum ISASIZE, you should use the REPORT option. The fastest initialization will be achieved if you specify an ISASIZE 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.

The REPORT option monitors the storage usage throughout the program, and prints the results at the end of execution. The values given and their meanings are described below.

ISASIZE SPECIFIED
This is the value specified. 0 is given if the ISASIZE option is not used.

LENGTH OF INITIAL STORAGE AREA (ISA)
This is the size that is acquired. It will differ from the value above if no value is specified, or if too small a value for the minimum requirements is specified.

AMOUNT OF PL/I STORAGE REQUIRED
This is the size of the ISA that gives fastest execution for that run of the program. Note, however, that this could waste storage if a large amount of storage is used only during part of program execution.

NUMBER OF GETMAINS
NUMBER OF FREEMAINS
This is the number of times storage outside the ISA is acquired. GETMAINS and FREEMAINS take time, and should be reduced when fast performance is required.

NUMBER OF GET NON-LIFO REQUESTS
NUMBER OF FREE NON-LIFO REQUESTS
These are the number of times storage is allocated for nonblock-dependent items such as BASED variables (as opposed to block-dependent variables). These values have little bearing on ISASIZE. For more information, refer to DOS PL/I Optimizing Compiler: Execution Logic.

The execution-time COUNT and FLOW options permit you to override the compiler COUNT and FLOW options. However, in normal use, it is just as simple to change the compiler options with a *PROCESS statement as it is to code the PLIXOPT string. The execution-time COUNT and FLOW options provide compatibility with CICS under OS/VS.

If the compiler option COUNT, or FLOW together with GOSTMT, is specified, the compiled program can produce both COUNT and FLOW output. The default is to produce COUNT output if COUNT is specified at compile time, and/or FLOW output if FLOW is specified. However, production of COUNT or FLOW output can be suppressed by specifying the CICS execution-time NOCOUNT or NOFLOW options, respectively. Also, FLOW output can be produced from a program compiled with COUNT and NOFLOW by specifying the CICS execution-time FLOW option; and COUNT output can be produced from a program compiled with NOCOUNT, FLOW, and GOSTMT by specifying the execution-time COUNT option.

If the FLOW compiler option is specified with NOGOSTMT, the compiled program can produce FLOW output, but not COUNT output. The default is to produce FLOW output. This can be suppressed by the CICS NOFLOW execution-time option. If the NOCOUNT and NOFLOW compiler options are specified, the CICS COUNT and FLOW execution-time options are ignored.

Note that the NOCOUNT and NOFLOW execution-time options only partially reduce the execution-time overhead of modules compiled with FLOW or COUNT. For best performance, the modules should be recompiled without FLOW and COUNT.

ERROR HANDLING

PL/I error handling is the same as under DOS, provided the STAE option is in effect. The only exception is that it is possible, under CICS, to override the automatic 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 that, as described above, is attached to the CPLI queue.

If the NOSTAE option is in effect, all hardware-detected interrupts and 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/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 the DOS STXIT macro in the CICS environment. If the STAE option is requested, a CICS DFHPC TYPE=SETXIT macro is issued by PL/I initialization. If a program check occurs, the CICS STXIT PC routine 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/VS itself or any other CICS transaction. It uses CICS/VS error handling; it does not override it.

If STAE is specified, CICS/VS control program services address the CICS version of the PL/I error handler as an exit routine to CICS/VS control program services. As such an exit routine, the PL/I error handler handles any CICS/VS abends, whether initiated by program checks or by software elsewhere in CICS, that occur in the PL/I program or associated CICS services. However, the PL/I error handler may not be able to handle abends that occur at a deeper level of the CICS system. It is your responsibility to see that such abends do not occur. Use of the DFHPC macro with an operand of TYPE=SETXIT or EXEC CICS HANDLE ABEND, while the STAE option is in effect, removes 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 want 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/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/VS (for example, a PL/I software interrupt raised ERROR). Backout may also be needed if 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 it available to program A.

To meet these requirements, the PL/I error handler under CICS/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 in your program, if present, 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, 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 does not keep the transaction from abending, and does 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 can occur and cause appropriate PL/I conditions to be raised. If not corrected in appropriate on-units, the software interrupt eventually causes the ERROR condition to be raised. If there is no ERROR on-unit, or if the program takes normal return from the ERROR on-unit, PL/I termination communicates to CICS/VS termination-in-error of the transaction by issuing a CICS/VS abend with abend code APLS. Thus, Dynamic Transaction Backout (for example) can proceed just as though CICS/VS had initiated the abend.

- When program A links to program B, and program B abends upon completion of CICS/VS, 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, the ERROR condition will be raised 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 possible to cope with computational interrupts, CONVERSION errors, and other non-I/O-related conditions using the same PL/I facilities used in programs executed directly under DOS.

For conditions associated with CICS/VS abends (including ASRA abends for program checks), CICS provides a facility (DFHPC

TYPE=SETXIT or EXEC CICS HANDLE ABEND;) to branch to either a program (external to the currently executing program) or to a routine (located somewhere within the current program). PL/I issues a DFHPC TYPE=SETXIT identifying the PL/I error handler as a program to establish linkage from CICS to the PL/I error handler, so use of this facility will necessarily destroy PL/I error handling. In PL/I programs, CICS/VS does not support a SETXIT identifying the label of a location in the current program. This is not really a restriction, because PL/I ON, SIGNAL, and REVERT statements give you all the facilities of PL/I to do so.

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 only exception to this rule.) Conditions detected by CICS/VS during the processing of your program are reflected via CICS-defined protocols. These are described in the CICS manuals.

In command-level programs, such conditions are reflected based on previously executed EXEC CICS HANDLE statements. The EXEC CICS HANDLE facility semantically resembles a PL/I on-unit of the form:

```
  ON condition GO TO label;
```

The HANDLE command can be coded wherever the ON ... GO TO ... statement can be coded. 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 not semantically identical to the ON ... GO TO ... statement; however, 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 the ERROR condition 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 an invalid 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, because 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. It may mean that a program check occurred in the PL/I error handler, and this, in turn, may mean that an error occurred that is so serious that it overwrote the PL/I environment, the PL/I error handler, or part of CICS itself.

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 maximum of 65512 permitted by CICS/VS. This error is caused by 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:

1. Termination is caused by the ERROR condition, and
2. 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/VS as an abend so that Dynamic Transaction Backout, and so on, can occur if necessary. Since there was no CICS/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) was intercepted by the PL/I error handler, but you were unable to resolve the condition. For instance, your program was terminated due to normal return from an on-unit. PL/I cannot re-issue the abend with code ASRA, because a program linked to this failing program would be abended with ASRA, which implies that a PSW and registers are supplied that permit fixup or retry while, in fact, the PSW is from a program no longer active, and the registers point to storage locations that are no longer meaningful.

APLX The total possible LIFO storage segments have been exhausted. Check the program for loops or increase the ISASIZE.

XXXX An abend is issued with the original abend code if termination was caused by the ERROR condition being raised with an abend code other than ASRA, and no IBMBEER module was included to cause user-specified action for the ERROR condition.

IBMBEER

IBMBEER's return code indicates whether a simple return or an abend is to be issued. IBMBEER is described in the section "The Abend Facility" in DOS PL/I Optimizing Compiler: Installation.

USE OF PLIDUMP

The CALL PLIDUMP statement is used to obtain a dump of storage areas in PL/I terms. Areas to be dumped can be specified via an options list in the same way as 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 actually executed. This means that CALL PLIDUMP statements may be included in production programs to be executed should unexpected errors arise.

The following options are available:

T	Trace of active procedures, etc.
NT	No trace
S	Stop execution
C	Continue execution
B	Produce a hexadecimal dump of PL/I control blocks (DSAs, PL/I TCA, etc.)
NB	No dump of PL/I blocks
K	Produce a hexadecimal dump of the TIOAS and TWA (CICS control blocks if they exist)
NK	No dump of CICS blocks

The default values are T, C, NB, and NK.

The dump information is built 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 produced by the CALL PLIDUMP statement has been transmitted. This means that no more than one transaction at any one time produces 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. For details about how a dump will be printed, contact your system programmer.

Because PLIDUMP does not print the program or its static storage, and there are many CICS/VS control blocks that it does not print, it may be appropriate to request a CICS dump in addition to PLIDUMP.

PLIDUMP copes with program checks that arise during its own execution; however, it is unable to cope with such program checks in the CICS/VS environment unless the program being dumped is executed with the STAE option in effect.

INTERLANGUAGE COMMUNICATION—OPTIONS ASSEMBLER

OPTIONS 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 information on the use of CICS/VS command-level facilities in an assembler language subroutine.

STORAGE AND CURRENTSTORAGE

The STORAGE and CURRENTSTORAGE builtin functions make the length of an item used for input or output available to the PL/I program. This is particularly useful with CICS, where functions often require the length of an argument as well as its address.

STORAGE and CURRENTSTORAGE are used with the command-level interface to eliminate having to count or compute PL/I aggregate lengths or specify length fields in the CICS commands.

PL/I PROGRAM TERMINATION

Most PL/I programs appear to terminate by returning from the main procedure, and may even appear to be a return to the Disk Operating System. In fact, it is a return to PL/I initialization/termination routines to perform various cleanup functions. In case of problems 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, for instance, not via a GOTO statement), the PL/I program terminates via PL/I termination facilities. A small percentage of PL/I programs terminate via a STOP or SIGNAL FINISH statement (although SIGNAL FINISH is a nonoperative statement unless a FINISH on-unit (even a null one) has been established). All of these, however, cause the program to terminate via PL/I termination facilities.

In the CICS/VS environment, PL/I programs terminate in any of the above ways, or they terminate via CICS/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 command-level 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, 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 the user inserts 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 with CICS. The long-range solution is to convert the program to use the command-level interface.

When PL/I program termination occurs via normal PL/I facilities, any requested FLOW, COUNT, or REPORT output is written to SYSPRINT, SYSPRINT is closed (if it's open), and the program returns control to CICS/VS. CICS/VS then frees all storage that the PL/I program acquired from CICS/VS via CICS GETMAINs issued by the PL/I library.

PL/I SHARED LIBRARY FOR CICS/DOS/VS

Shared library support for CICS/DOS/VS is supplied by DOS PL/I. It is not a generalized shared library facility, it is not available to non-CICS/VS programs, and the user has no control over its contents.

If shared library support is installed, it is link-edited as an integral part of DFHSAP, increasing the size of DFHSAP from 7K to 8K bytes to about 19K bytes. It contains:

- IBMDPSL - DOS shared library bootstrap
- IBMBAMM - Structure mapping, sometimes used for STORAGE and CURRENTSTORAGE BIFs
- PL/I bit manipulation routines
- PL/I conversion library

You must supply the linkage editor control statement "INCLUDE PLISHRE". This supplies a bootstrap module, IBMDPSR, which is link-edited into your program to resolve shared library modules. CICS/VS initialization supplies addressability to the shared library. As a rule, DOS/VS PL/I-CICS/VS users likely to have four or more PL/I transactions active concurrently benefit from this shared library support.

USING THE CICS FACILITIES

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 DFHPL1I, which acts as the entry point to the program and must be link-edited with the application program, is provided. Also, the CICS loader stores certain addresses within this interface module, and the loader assumes it is positioned at the head of the load module.

An INCLUDE card for DHFPL1I must exist immediately after the phase name. The link-edited output should be checked to ensure that DFHPL1I is the entry point and DFHIBM is the first CSECT. For example, to compile a program and produce a module called PROG1, use:

```
//OPTION CATAL
  PHASE PROG1,*
  INCLUDE DFHPL1I
/*
//EXEC PLIOPT, SIZE=64K
//EXEC LNKEDT
/*
```

assuming, earlier in the same job, the PL/I source has been preprocessed by the CICS preprocessor.

If the shared library is to be used, an INCLUDE PLISHRE card should follow the INCLUDE DFHPL1I card.

APPENDIX A. PROGRAMMING EXAMPLE

The example illustrates page by page all the listings produced by the optimizing compiler and the linkage editor. The listings are described in Chapter 4 on page 14 and Chapter 5 on page 43.

The example program illustrates the use of the preprocessor %INCLUDE statement to incorporate a PL/I DECLARE statement from the source statement library. The program contains statements to initialize a two-dimensional table with the values 1 through 12 in each column, and then proceeds to calculate and print the products of each row and column of this table. The source statements for this example are given in Figure 102.

```
// JOB FIGA1
// OPTION LINK
// EXEC PLIOPT,SIZE=64K
* PROCESS MACROS,INSOURCE,AGGREGATE< ATTRIBUTES,ESD,NEST,OFFSET,
  OPT(TIME),STORAGE,XREF,LIST;

 P: PROC OPTIONS(MAIN) REORDER;
    %INCLUDE DCLSTMT;                         /* INCLUDE DECLARATIONS */

    DO I = 1 TO 2;                            /* INITIALIZE TABLE */
       DO J = 1 TO 12;
          A(J,I) = J;
       END;
    END;

    DO J = 1 TO 2;                            /* CALCULATE COLUMN PRODUCTS */
       COLPROD(J) =1;
       DO I = 1 TO 12;
          COLPROD(J) = COLPROD(J) * A(I,J);
       END;
    END;

    DO I = 1 TO 12;                           /* CALCULATE ROW PRODUCTS */
          ROWPROD(I) = A(I,1) * A(I,2);
    END;

    PUT SKIP(2) LIST ('PRODUCTS OF ROWS');    /* PRINT RESULTS */
    PUT SKIP(2) DATA (ROWPROD);
    PUT SKIP(2) LIST ('PRODUCTS OF COLUMNS');
    PUT SKIP (2) DATA (COLPROD);
 END P;
/*
// EXEC LNKEDT
// EXEC ,SIZE=64K
/&
```

Figure 102. Example Program Source Statements

CONTENTS OF LISTINGS

Page 1:	JOB, OPTION, and EXEC statements as they appear in the input.
Page 2:	First page of compiler listing, showing the list of options applicable to the compilation as well as those that were given in the PROCESS statement. Note that the size option gives the actual amount of storage available for compilation.

Page 3:	The listing of the input to the preprocessor. Lines 1 to 26 contain a PL/I source program that is to be modified by the preprocessor. Lines 27 and 28 contain the included statement read from BOOKNAME P.DCLSTMT. The message indicating that no errors were detected during preprocessing follows the reproduction of the included source statement module.
Page 4:	The same program created by the preprocessor as listed by the compiler.
Page 5:	The attribute and cross-reference table.
Page 6:	The aggregate length table.
Page 7:	The storage requirement table.
Page 8:	The external symbol dictionary and compilation statistics.
Page 9,10:	The static internal storage map.
Page 11:	The variable storage map.
Page 12:	The table of offsets and statement numbers.
Pages 13-16:	The object listing.
Page 17:	The compiler diagnostic message produced for this compilation. The time taken for the compilation and the size and number of records transmitted to the spill file during compilation are also given.
Page 18:	An EXEC statement to invoke the linkage editor.
Page 19:	The diagnostic listing produced by the linkage editor. "ACTION TAKEN" indicates that the map option of the ACTION statement has been assumed by default. "LIST" means that the listed feature or control statement has been used. In this example, the AUTOLINK feature has been used for the name relocatable object modules.
Pages 20,21:	The linkage editor map, showing the relative storage locations in the executable program phase of all the control sections created by the compiler or incorporated from the PL/I resident library. The listing of unreferenced symbols and the reference to unresolved address constants apply to external references that cannot be matched with a corresponding entry point. These are, in general, weak external references (WXTRN) that are not needed in the program.
Page 22:	The EXEC statement to invoke the executable program phase created in the preceding link-editing step.
Page 23:	Listed output from the execution of the compiled and link-edited program.

```
// JOB RFP214AB                                                                              ①
// OPTION LINK
// EXEC PLIOPT,SIZE=64K
```

```
DOS PL/I OPTIMIZING COMPILER            VERSION 1 RELEASE 5.0          TIME: 18.24.39    DATE: 26 AUG 76          PAGE   1

OPTIONS SPECIFIED   .
                                                                                                                    ②
     * PROCESS MACRO,INSOURCE,AGGREGATE,ATTRIBUTES,ESD,MAP,NEST,OFFSET,       00120000

       OPT(TIME),STORAGE,XREF,LIST;                                           00130000

OPTIONS USED

AGGREGATE      NOCOUNT        ATTRIBUTES(SHORT)
COMPILE        NODECK         CHARSET(60,EBCDIC)
DYNBUF         NOFLOW         FLAG(W)
ESD            NOLIMSCONV     LINECOUNT(60)
GOSTMT         NOMARGINI      MARGINS(2,72,1)
INCLUDE        NOMDECK        OPTIMIZE(TIME)
INSOURCE                      SIZE(389000)
LINK                          WORKFILE(3330)
LIST                          XREF(SHORT)
MACRO
MAP
NEST
OFFSET
OPTIONS
SOURCE
STORAGE
SYNTAX
```

```
PL/I OPTIMIZING COMPILER            P: PROC OPTIONS(MAIN) REORDER;                                              PAGE   2

                      PREPROCESSOR INPUT
                                                                                                                  ③
 LINE
    1          P: PROC OPTIONS(MAIN) REORDER;                                      00140000
    2             /* INCLUDE DECLARATIONS */                                       00150000
    3             %INCLUDE DCLSTMT;                                                00160001
    4             /* INITIALIZE TABLE */                                           00170000
    5             DO I = 1 TO 2;                                                   00180000
    6                DO J = 1 TO 12;                                               00190000
    7                   A(J,I) = J;                                                00200000
    8                   END;                                                       00210000
    9                END;                                                          00220000
   10             /* CALCULATE COLUMN PRODUCTS */                                  00230000
   11             DO J = 1 TO 2;                                                   00240000
   12                COLPROD(J) = 1;                                               00250000
   13                DO I = 1 TO 12;                                               00260000
   14                   COLPROD(J) = COLPROD(J) * A(I,J);                          00270000
   15                   END;                                                       00280000
   16                END;                                                          00290000
   17             /* CALCULATE ROW PRODUCTS */                                     00300000
   18             DO I = 1 TO 12;                                                  00310000
   19                  ROWPROD(I) = A(I,1) * A(I,2);                               00320000
   20                END;                                                          00330000
   21             /* PRINT RESULTS */                                              00340000
   22             PUT SKIP(2) LIST ('PRODUCTS OF ROWS');                           00350000
   23             PUT SKIP(2) DATA (ROWPROD);                                      00360000
   24             PUT SKIP(2) LIST ('PRODUCTS OF COLUMNS');                        00370000
   25             PUT SKIP(2) DATA (COLPROD);                                      00380000
   26             END P;                                                           00390000


   INCLUDED TEXT FOLLOWS FROM BOOKNAME =  P.DCLSTMT

   27                 DCL (ROWPROD(12),COLPROD(2),A(12,2))                         00040000
   28                      FIXED DECIMAL (5);                                      00050000


PL/I OPTIMIZING COMPILER            P: PROC OPTIONS(MAIN) REORDER;                                              PAGE   3

NO MESSAGES OF SEVERITY W AND ABOVE PRODUCED BY THE PREPROCESSOR



MESSAGES SUPPRESSED BY THE FLAG OPTION:  1 I.
```

```
PL/I OPTIMIZING COMPILER          P: PROC OPTIONS(MAIN) REORDER;                            PAGE   4

                   MACRO AND SOURCE LISTING                                                      ④

    STMT LEV NT                                                                     R

      1      0      P: PROC OPTIONS(MAIN) REORDER;                                      1
                       /* INCLUDE DECLARATIONS */                                       2
      2   1  0              DCL (ROWPROD(12),COLPROD(2),A(12,2))                        27
                                FIXED DECIMAL (5);                                      28
                       /* INITIALIZE TABLE */                                           4
      3   1  0         DO I = 1 TO 2;                                                   5
      4   1  1            DO J = 1 TO 12;                                               6
      5   1  2               A(J,I) = J;                                                7
      6   1  2               END;                                                       8
      7   1  1            END;                                                          9
                       /* CALCULATE COLUMN PRODUCTS */                                  10
      8   1  0         DO J = 1 TO 2;                                                   11
      9   1  1            COLPROD(J) = 1;                                               12
     10   1  1            DO I = 1 TO 12;                                               13
     11   1  2               COLPROD(J) = COLPROD(J) * A(I,J);                          14
     12   1  2               END;                                                       15
     13   1  1            END;                                                          16
                       /* CALCULATE ROW PRODUCTS */                                     17
     14   1  0         DO I = 1 TO 12;                                                  18
     15   1  1              ROWPROD(I) = A(I,1) * A(I,2);                               19
     16   1  1            END;                                                          20
                       /* PRINT RESULTS */                                              21
     17   1  0         PUT SKIP(2) LIST ('PRODUCTS OF ROWS');                           22
     18   1  0         PUT SKIP(2) DATA (ROWPROD);                                      23
     19   1  0         PUT SKIP(2) LIST ('PRODUCTS OF COLUMNS');                        24
     20   1  0         PUT SKIP(2) DATA (COLPROD);                                      25
     21   1  0         END P;                                                           26
```

```
PL/I OPTIMIZING COMPILER          P: PROC OPTIONS(MAIN) REORDER;                            PAGE   5

                                             ATTRIBUTE AND CROSS-REFERENCE TABLE (SHORT)
                                                                                                 ⑤
DCL NO.     IDENTIFIER                    ATTRIBUTES AND REFERENCES

2           A                             (12,2) AUTOMATIC ALIGNED DECIMAL FIXED (5,0)
                                          5,11,15,15

2           COLPROD                       (2) AUTOMATIC ALIGNED DECIMAL FIXED (5,0)
                                          9,11,11,20

********    I                             AUTOMATIC ALIGNED BINARY FIXED (15,0)
                                          3,3,5,10,10,11,14,14,15,15,15

********    J                             AUTOMATIC ALIGNED BINARY FIXED (15,0)
                                          4,4,5,5,8,8,9,11,11,11

2           ROWPROD                       (12) AUTOMATIC ALIGNED DECIMAL FIXED (5,0)
                                          15,18

********    SYSPRINT                      EXTERNAL FILE PRINT
                                          17,18,19,20
```

```
PL/I OPTIMIZING COMPILER          P: PROC OPTIONS(MAIN) REORDER;                                          PAGE   6

                                      AGGREGATE LENGTH TABLE                                                   ⑥
DCL NO.   IDENTIFIER                              LVL      DIMS       OFFSET  ELEMENT    TOTAL
                                                                               LENGTH.    LENGTH.

2         A                                                  2                      3         72

2         COLPROD                                            1                      3          6

2         ROWPROD                                            1                      3         36

                                                              SUM OF CONSTANT LENGTHS        114
```

```
PL/I OPTIMIZING COMPILER          P: PROC OPTIONS(MAIN) REORDER;                                          PAGE   7

                      STORAGE REQUIREMENTS
                                                                                                           ⑦
BLOCK, SECTION OR STATEMENT       TYPE                  LENGTH    (HEX)    DSA SIZE    (HEX)

******P1                          PROGRAM CSECT            744      2E8
******P2                          STATIC CSECT             500      1F4
P                                 PROCEDURE BLOCK          742      2E6         464      1D0
```

```
PL/I OPTIMIZING COMPILER          P: PROC OPTIONS(MAIN) REORDER;                                          PAGE   8

                      EXTERNAL SYMBOL DICTIONARY
                                                                                                           ⑧
           SYMBOL         TYPE      ID       ADDR      LENGTH

          PLISTART         SD      0001     000000     000010
          ******P1         SD      0002     000000     0002E8
          ******P2         SD      0003     000000     0001F4
          PLITABS          WX      0004     000000
          PLIFLOW          WX      0005     000000
          PLICOUNT         WX      0006     000000
          IBMBPIRA         ER      0007     000000
          PLIMAIN          SD      0008     000000     000008
          IBMBSDOA         ER      0009     000000
          IBMBSIOA         ER      000A     000000
          IBMBSDOA         ER      000B     000000
          IBMBCACA         ER      000C     000000
          IBMBCWDH         ER      000D     000000
          IBMBOCLA         ER      000E     000000
          IBMBOCLC         WX      000F     000000
          IBMBSDOB         WX      0010     000000
          IBMBSIOE         WX      0011     000000
          IBMBSLOA         ER      0012     000000
          IBMBSPLA         ER      0013     000000
          IBMBSDOT         WX      0014     000000
          IBMBSXCA         WX      0015     000000
          IBMBSXCB         WX      0016     000000
          IBMBSIST         WX      0017     000000
          IBMDSTFA         ER      0018     000000
          IBMBOCLB         ER      0019     000000
          IJJFCBZD         ER      001A     000000
          P                LD               000004
          SYSPINT          SD      001B     000000     0001A8
```

```
PL/I OPTIMIZING COMPILER              P: PROC OPTIONS(MAIN) REORDER;                                      PAGE   9

                        STATIC INTERNAL STORAGE MAP                      C4000000
                                                         0000FC  810001010000003A        SYMBOL TABLE..COLPROD
000000  C00001F0                PROGRAM ADCON                    000000C000000000
000004  00000004                PROGRAM ADCON                    0007C3D6D3D7D9D6                          ⑨
000008  0000005A                PROGRAM ADCON                    C4000000
00000C  00000078                PROGRAM ADCON
000010  00000078                PROGRAM ADCON
000014  00000000                A..IBMBCACA
000018  00000000                A..IBMBCWDH
00001C  00000000                A..IBMBOCLA                                          STATIC EXTERNAL CSECTS
000020  00000000                A..IBMBOCLC
000024  00000000                A..IBMBSDOB
000028  00000000                A..IBMBSIOE
00002C  00000000                A..IBMBSLOA              000000  0000000000000000        FCB
000030  00000000                A..IBMBSPLA                      0000000000000000
000034  00000000                A..IBMBSDOT                      000000700000007C
000038  2000                    DED                              000000B800000000
00003A  04040580                DED..ROWPROD                     4040000041201000
00003E  0002                    CONSTANT                         8080001002000000
000040  0001                    CONSTANT                         0000E20000000000
000042  0009                    CONSTANT                         0000000000000000
000044  000C                    CONSTANT                         0000008000000000
000046  0006                    CONSTANT                         0000000000000000
000048  0003                    CONSTANT                         0000000000000000
00004A  004B                    CONSTANT                         0000000000000000
00004C  0048                    CONSTANT                         0000000000000000
00004E  0024                    CONSTANT                         0000000000000000
000050  000000BB00100000        LOCATOR                          0008E2E8E2D7D9C9
000058  0000000000000078        LOCATOR..ROWPROD                 D5E30000
000060  000000CB00130000        LOCATOR                  00007C  00000000020000AC        ENVB
000068  0000000000000084        LOCATOR..COLPROD                 010000B0020000AC
000070  91E091E0                CONSTANT                         020000AC020000AC
000074  00000002                CONSTANT                         020000AC020000AC
000078  0000000300000003        DESCRIPTOR                       020000AC020000AC
        000C0001                                                 020000AC020000AC
000084  0000000300000003        DESCRIPTOR               0000AC  00000000                ENVB CONSTANT
        00020001                                         0000B0  00000079                ENVB CONSTANT
000090  00000000                A..FCB                   0000B8  0000800008000003        DTF (CONSTANT PART)
000094  00000000                A..FCB                   0000C0  00000110                DTF (VARIABLE PART)
000098  00000000                A..TEMP                  0000C4  0000000000000000        DTF (CONSTANT PART)
00009C  80000074                A..CONSTANT                      3380E2E8E2D7D9C9
0000A0  00000000                A..TEMP                          D500000000000000
0000A4  000000E0                A..SYMTAB                        00000800002020F3
0000A8  80000000                A..TEMP                          2410610880000000
0000AC  00000000                A..TEMP                          0000000000000000
0000B0  000000FC                A..SYMTAB                        0000FF0000000000
0000B4  80000000                A..TEMP                          0000000013000020
0000B8  00001C                  CONSTANT                         0000000000000079
0000BB  D7D9D6C4E4C3E3E2        CONSTANT                         47000000
        40D6C640D9D6E6E2                                 000110  070000F2                DTF (VARIABLE PART)
0000CB  D7D9D6C4E4C3E3E2        CONSTANT                 000114  40000006                DTF (CONSTANT PART)
        40D6C640C3D6D3E4                                 000118  310000F4                DTF (VARIABLE PART)
        D4D5E2                                           00011C  40000005                DTF (CONSTANT PART)
0000DE                                                   000120  08000118                DTF (VARIABLE PART)
0000E0  810001010000003A        SYMBOL TABLE..ROWPROD    000124  20000001                DTF (CONSTANT PART)
        000000B800000000                                 000128  1D000104                DTF (VARIABLE PART)
        0007D9D6E6D7D9D6                                 00012C  A00000080510610C        DTF (CONSTANT PART)
```

```
PL/I OPTIMIZING COMPILER           P: PROC OPTIONS(MAIN) REORDER;                              PAGE  10

        60000079
000138  310000F4             DTF (VARIABLE PART)                                                 10
00013C  40000005             DTF (CONSTANT PART)
000140  08000138             DTF (VARIABLE PART)
000144  20000001             DTF (CONSTANT PART)
000148  1E000128             DTF (VARIABLE PART)
00014C  3000008100000000     DTF (CONSTANT PART)
        0000000000000000
        0000000000000000
        0000000000000000
        0000000000000000
        0000000000000000
        0000000000000000
        0000000000000000
        0000000000000000
        0000000000000000
        0000000047FF001C
        47FF001C
```

```
PL/I OPTIMIZING COMPILER           P: PROC OPTIONS(MAIN) REORDER;                              PAGE  11

                                 VARIABLE STORAGE MAP
                                                                                                 11
IDENTIFIER                           LEVEL    OFFSET     (HEX)    CLASS      BLOCK

ROWPROD                                  1       216        D8    AUTO       P
COLPROD                                  1       204        CC    AUTO       P
A                                        1       252        FC    AUTO       P
I                                        1       200        C8    AUTO       P
J                                        1       202        CA    AUTO       P
```

```
PL/I OPTIMIZING COMPILER           P: PROC OPTIONS(MAIN) REORDER;                              PAGE  12

             TABLES OF OFFSETS AND STATEMENT NUMBERS                                             12


             WITHIN PROCEDURE P

OFFSET (HEX)      0     74     7C     84     84     9A     A8     A8     AA     B2     B2     B2     BE     D2     D2
STATEMENT NO.     1      3      5      3      4      5      6      5      6      4      7      5      7      3      8

OFFSET (HEX)     DA     E2     F2     F2    100    118    132    136    136    136    142    15A    16E    16E    176
STATEMENT NO.     9     11      8      9     10     11     12     10     13      9     11     13      8     14     15

OFFSET (HEX)    182    18A    1A4    1A4    1A6    1AA    1AA    1DC    23A    26C    2D2
STATEMENT NO.    14     15     16     15     16     14     17     18     19     20     21
```

```
PL/I OPTIMIZING COMPILER           P: PROC OPTIONS(MAIN) REORDER;                                  PAGE  13

  OBJECT LISTING


* STATEMENT NUMBER  1
000000                                 DC    C'  P'
000003                                 DC    AL1(1)

* PROCEDURE                                  P

* REAL ENTRY
000004  90 EC D 00C                    STM   14,12,12(13)
000008  47 F0 F 014                    B     *+16
00000C  00000000                       DC    A(STMT. NO. TABLE)
000010  000001D0                       DC    F'464'
000014  00000000                       DC    A(STATIC CSECT)
000018  58 30 F 010                    L     3,16(0,15)
00001C  58 10 D 04C                    L     1,76(0,13)
000020  58 00 F 00C                    L     0,12(0,15)
000024  1E 01                          ALR   0,1
000026  55 00 C 00C                    CL    0,12(0,12)
00002A  47 D0 F 030                    BNH   *+10
00002E  58 F0 C 074                    L     15,116(0,12)
000032  05 EF                          BALR  14,15
000034  58 E0 D 048                    L     14,72(0,13)
000038  18 F0                          LR    15,0
00003A  90 E0 1 048                    STM   14,0,72(1)
00003E  50 D0 1 004                    ST    13,4(0,1)
000042  41 D1 0 000                    LA    13,0(1,0)
000046  50 50 D 058                    ST    5,88(0,13)
00004A  92 80 D 000                    MVI   0(13),X'80'
00004E  92 24 D 001                    MVI   1(13),X'24'
000052  D2 03 D 054 3 070              MVC   84(4,13),112(3)
000058  05 20                          BALR  2,0

* PROLOGUE BASE
00005A  D2 07 D 0B8 3 058              MVC   LOCATOR..VO..ROWPR
                                             OD(8),88(3)
000060  41 E0 D 0D8                    LA    14,216(0,13)
000064  50 E0 D 0B8                    ST    14,LOCATOR..VO..RO
                                             WPROD
000068  D2 07 D 0C0 3 068              MVC   LOCATOR..VO..COLPR
                                             OD(8),104(3)
00006E  41 F0 D 0CC                    LA    15,204(0,13)
000072  50 F0 D 0C0                    ST    15,LOCATOR..VO..CO
                                             LPROD
000076  05 20                          BALR  2,0

* PROCEDURE BASE


* STATEMENT NUMBER  3
000078  48 50 3 040                    LH    5,64(0,3)
00007C  40 50 D 0C8                    STH   5,I

* INITIALIZATION CODE FOR OPTIMIZED LOOP FOLLOWS
```

```
* CODE MOVED FROM STATEMENT NUMBER 5
000080  48 E0 3 042                LH    14,66(0,3)
000084  50 E0 D 160                ST    14,352(0,13)

* CONTINUATION OF STATEMENT NUMBER  3                   13
000088                     CL.2    EQU   *

* STATEMENT NUMBER  4
000088  48 B0 3 044                LH    11,68(0,3)
00008C  48 A0 3 040                LH    10,64(0,3)
000090  40 A0 D 0CA                STH   10,J

* INITIALIZATION CODE FOR OPTIMIZED LOOP FOLLOWS

* CALCULATION OF COMMONED EXPRESSION FOLLOWS
000094  58 80 D 160                L     8,352(0,13)

* END OF COMMON CODE
000098  48 40 3 046                LH    4,70(0,3)
00009C  18 5A                      LR    5,10
00009E                     CL.4    EQU   *


* STATEMENT NUMBER  5
00009E  4E 50 D 090                CVD   5,WKSP.1+24
0000A2  18 7D                      LR    7,13
0000A4  1A 78                      AR    7,8
0000A6  D2 02 7 0F3 D 095          MVC   VO..A(3),WKSP.1+29


* STATEMENT NUMBER  6

* METHOD OR ORDER OF CALCULATING EXPRESSIONS CHANGED

* CODE MOVED FROM STATEMENT NUMBER 5
0000AC  1A 84                      AR    8,4

* CONTINUATION OF STATEMENT NUMBER  6
0000AE  87 5A 2 026                BXLE  5,10,CL.4
0000B2  40 50 D 0CA                STH   5,J

* CODE MOVED FROM STATEMENT NUMBER 4


* STATEMENT NUMBER  7

* METHOD OR ORDER OF CALCULATING EXPRESSIONS CHANGED

* CODE MOVED FROM STATEMENT NUMBER 5
0000B6  58 E0 D 160                L     14,352(0,13)
0000BA  4A E0 3 048                AH    14,72(0,3)
0000BE  50 E0 D 160                ST    14,352(0,13)

* CONTINUATION OF STATEMENT NUMBER  7
0000C2  48 50 D 0C8                LH    5,I
0000C6  4A 50 3 040                AH    5,64(0,3)
```

```
0000CA  40 50 D 0C8                 STH   5,I
0000CE  49 50 3 03E                 CH    5,62(0,3)
0000D2  47 C0 2 010                 BNH   CL.2

* CODE MOVED FROM STATEMENT NUMBER 3


* STATEMENT NUMBER  8
0000D6  48 50 3 040                 LH    5,64(0,3)
0000DA  40 50 D 0CA                 STH   5,J

* INITIALIZATION CODE FOR OPTIMIZED LOOP FOLLOWS

* CODE MOVED FROM STATEMENT NUMBER 9
0000DE  48 40 3 048                 LH    4,72(0,3)
0000E2  50 40 D 168                 ST    4,360(0,13)

* CODE MOVED FROM STATEMENT NUMBER 11
0000E6  48 80 3 042                 LH    8,66(0,3)
0000EA  50 80 D 16C                 ST    8,364(0,13)
0000EE  48 90 3 04A                 LH    9,74(0,3)
0000F2  50 90 D 170                 ST    9,368(0,13)

* CONTINUATION OF STATEMENT NUMBER  8
0000F6                     CL.6     EQU   *


* STATEMENT NUMBER  9
0000F6  58 70 D 168                 L     7,360(0,13)
0000FA  18 4D                       LR    4,13
0000FC  1A 47                       AR    4,7
0000FE  D2 02 4 0C9 3 0B8           MVC   VO..COLPROD(3),184
                                          (3)


* STATEMENT NUMBER  10
000104  48 90 3 040                 LH    9,64(0,3)
000108  40 90 D 0C8                 STH   9,I

* INITIALIZATION CODE FOR OPTIMIZED LOOP FOLLOWS

* CALCULATION OF COMMONED EXPRESSION FOLLOWS
00010C  58 E0 D 16C                 L     14,364(0,13)

* END OF COMMON CODE
000110  58 B0 D 170                 L     11,368(0,13)
000114  48 A0 3 046                 LH    10,70(0,3)
000118  18 5E                       LR    5,14
00011A  18 87                       LR    8,7
00011C                     CL.8     EQU   *


* STATEMENT NUMBER  11
00011C  18 4D                       LR    4,13
00011E  1A 48                       AR    4,8
000120  18 9D                       LR    9,13
000122  1A 95                       AR    9,5
```

```
000124  F8 52 D 098 4 0C9           ZAP   WKSP.1+32(6),VO..C
                                          OLPROD(3)
00012A  FC 52 D 098 9 0F3           MP    WKSP.1+32(6),VO..A
                                          (3)
000130  D2 02 4 0C9 D 09B           MVC   VO..COLPROD(3),WKS
                                          P.1+35
                                                        14
* STATEMENT NUMBER  12
000136  87 5A 2 0A4                 BXLE  5,10,CL.8

* CODE MOVED FROM STATEMENT NUMBER 10


* STATEMENT NUMBER  13

* METHOD OR ORDER OF CALCULATING EXPRESSIONS CHANGED

* CODE MOVED FROM STATEMENT NUMBER 9
00013A  58 E0 D 168                 L     14,360(0,13)
00013E  4A E0 3 048                 AH    14,72(0,3)
000142  50 E0 D 168                 ST    14,360(0,13)

* CODE MOVED FROM STATEMENT NUMBER 11
000146  58 80 D 16C                 L     8,364(0,13)
00014A  4A 80 3 048                 AH    8,72(0,3)
00014E  50 80 D 16C                 ST    8,364(0,13)
000152  58 40 D 170                 L     4,368(0,13)
000156  4A 40 3 048                 AH    4,72(0,3)
00015A  50 40 D 170                 ST    4,368(0,13)

* CONTINUATION OF STATEMENT NUMBER  13
00015E  48 50 D 0CA                 LH    5,J
000162  4A 50 3 040                 AH    5,64(0,3)
000166  40 50 D 0CA                 STH   5,J
00016A  49 50 3 03E                 CH    5,62(0,3)
00016E  47 C0 2 07E                 BNH   CL.6

* CODE MOVED FROM STATEMENT NUMBER 8


* STATEMENT NUMBER  14
000172  48 70 3 040                 LH    7,64(0,3)
000176  40 70 D 0C8                 STH   7,I

* INITIALIZATION CODE FOR OPTIMIZED LOOP FOLLOWS

* CODE MOVED FROM STATEMENT NUMBER 15
00017A  48 40 3 048                 LH    4,72(0,3)
00017E  48 E0 3 046                 LH    14,70(0,3)
000182  48 90 3 04C                 LH    9,76(0,3)

* CONTINUATION OF STATEMENT NUMBER  14
000186  18 64                       LR    6,4
000188  18 5E                       LR    5,14
00018A  18 B9                       LR    11,9
00018C  18 AE                       LR    10,14
```

```
PL/I OPTIMIZING COMPILER          P: PROC OPTIONS(MAIN) REORDER;                              PAGE  15

00018E                    CL.10   EQU   *

* STATEMENT NUMBER  15
00018E  18 7D                     LR    7,13
000190  1A 75                     AR    7,5
000192  18 9D                     LR    9,13
000194  1A 94                     AR    9,4
000196  F8 52 D 098 7 0F6         ZAP   WKSP.1+32(6),VO..A
                                        +3(3,7)
00019C  FC 52 D 098 7 0F9         MP    WKSP.1+32(6),VO..A
                                        +6(3,7)
0001A2  D2 02 9 0D5 D 09B         MVC   VO..ROWPROD(3),WKS
                                        P.1+35

* STATEMENT NUMBER  16

* METHOD OR ORDER OF CALCULATING EXPRESSIONS CHANGED

* CODE MOVED FROM STATEMENT NUMBER 15
0001A8  1A 46                     AR    4,6

* CONTINUATION OF STATEMENT NUMBER  16
0001AA  87 5A 2 116               BXLE  5,10,CL.10

* CODE MOVED FROM STATEMENT NUMBER 14


* STATEMENT NUMBER  17
0001AE  41 F0 D 1A8               LA    15,424(0,13)
0001B2  50 F0 3 098               ST    15,152(0,3)
0001B6  50 F0 D 188               ST    15,392(0,13)
0001BA  18 1F                     LR    1,15
0001BC  92 40 D 1B9               MVI   441(13),X'40'
0001C0  41 10 3 094               LA    1,148(0,3)
0001C4  58 F0 3 028               L     15,A..IBMBSIOE
0001C8  05 EF                     BALR  14,15
0001CA  41 E0 3 050               LA    14,80(0,3)
0001CE  41 F0 3 038               LA    15,56(0,3)
0001D2  58 10 D 188               L     1,392(0,13)
0001D6  90 EF 1 000               STM   14,15,0(1)
0001DA  58 F0 3 02C               L     15,A..IBMBSLOA
0001DE  05 EF                     BALR  14,15


* STATEMENT NUMBER  18
0001E0  41 E0 D 1A8               LA    14,424(0,13)
0001E4  50 E0 3 098               ST    14,152(0,3)
0001E8  50 E0 D 188               ST    14,392(0,13)
0001EC  18 1E                     LR    1,14
0001EE  92 80 D 1B9               MVI   441(13),X'80'
0001F2  92 01 D 1BA               MVI   442(13),X'01'
0001F6  41 10 3 094               LA    1,148(0,3)
0001FA  58 F0 3 028               L     15,A..IBMBSIOE
0001FE  05 EF                     BALR  14,15
000200  48 80 3 040               LH    8,64(0,3)
```

```
* INITIALIZATION CODE FOR OPTIMIZED LOOP FOLLOWS                ⑮
000204  48 E0 3 048                       LH    14,72(0,3)
000208  48 40 3 04E                       LH    4,78(0,3)
00020C  18 B4                             LR    11,4
00020E  18 5E                             LR    5,14
000210  18 AE                             LR    10,14
000212                            CL.13   EQU   *
000212  41 70 D 1A8                       LA    7,424(0,13)
000216  50 70 3 0A0                       ST    7,160(0,3)
00021A  41 75 D 0D5                       LA    7,VO..ROWPROD(5)
00021E  50 70 3 0A8                       ST    7,168(0,3)
000222  96 80 3 0A8                       OI    168(3),X'80'
000226  41 10 3 0A0                       LA    1,160(0,3)
00022A  58 F0 3 024                       L     15,A..IBMBSDOB
00022E  05 EF                             BALR  14,15
000230  87 5A 2 19A                       BXLE  5,10,CL.13
000234  58 10 D 188                       L     1,392(0,13)
000238  58 F0 3 034                       L     15,A..IBMBSDOT
00023C  05 EF                             BALR  14,15

* STATEMENT NUMBER  19
00023E  41 E0 D 1A8                       LA    14,424(0,13)
000242  50 E0 3 098                       ST    14,152(0,3)
000246  50 E0 D 188                       ST    14,392(0,13)
00024A  18 1E                             LR    1,14
00024C  92 40 D 1B9                       MVI   441(13),X'40'
000250  41 10 3 094                       LA    1,148(0,3)
000254  58 F0 3 028                       L     15,A..IBMBSIOE
000258  05 EF                             BALR  14,15
00025A  41 E0 3 060                       LA    14,96(0,3)
00025E  41 F0 3 038                       LA    15,56(0,3)
000262  58 10 D 188                       L     1,392(0,13)
000266  90 EF 1 000                       STM   14,15,0(1)
00026A  58 F0 3 02C                       L     15,A..IBMBSLOA
00026E  05 EF                             BALR  14,15


* STATEMENT NUMBER  20
000270  41 E0 D 1A8                       LA    14,424(0,13)
000274  50 E0 3 098                       ST    14,152(0,3)
000278  50 E0 D 188                       ST    14,392(0,13)
00027C  18 1E                             LR    1,14
00027E  92 80 D 1B9                       MVI   441(13),X'80'
000282  92 01 D 1BA                       MVI   442(13),X'01'
000286  41 10 3 094                       LA    1,148(0,3)
00028A  58 F0 3 028                       L     15,A..IBMBSIOE
00028E  05 EF                             BALR  14,15
000290  48 E0 3 040                       LH    14,64(0,3)

* INITIALIZATION CODE FOR OPTIMIZED LOOP FOLLOWS
000294  50 A0 D 19C                       ST    10,412(0,13)
000298  48 40 3 046                       LH    4,70(0,3)
00029C  48 A0 3 048                       LH    10,72(0,3)
0002A0  58 50 D 19C                       L     5,412(0,13)
0002A4  18 B4                             LR    11,4
```

```
0002A6                      CL.12   EQU    *
0002A6  50 50 D 19C                 ST     5,412(0,13)
0002AA  41 70 D 1A8                 LA     7,424(0,13)                                          (16)
0002AE  50 70 3 0AC                 ST     7,172(0,3)
0002B2  41 75 D 0C9                 LA     7,VO..COLPROD(5)
0002B6  50 70 3 0B4                 ST     7,180(0,3)
0002BA  96 80 3 0B4                 OI     180(3),X'80'
0002BE  41 10 3 0AC                 LA     1,172(0,3)
0002C2  58 F0 3 024                 L      15,A..IBMBSDOB
0002C6  05 EF                       BALR   14,15
0002C8  87 5A 2 22E                 BXLE   5,10,CL.12
0002CC  58 10 D 188                 L      1,392(0,13)
0002D0  58 F0 3 034                 L      15,A..IBMBSDOT
0002D4  05 EF                       BALR   14,15


* STATEMENT NUMBER  21
0002D6  18 0D                       LR     0,13
0002D8  58 D0 D 004                 L      13,4(0,13)
0002DC  58 E0 D 00C                 L      14,12(0,13)
0002E0  98 2C D 01C                 LM     2,12,28(13)
0002E4  05 1E                       BALR   1,14

* END PROCEDURE
0002E6  07 07                       NOPR   7

* END PROGRAM
```

listing page 16.

```
PL/I OPTIMIZING COMPILER            P: PROC OPTIONS(MAIN) REORDER;                                    PAGE  17

COMPILER DIAGNOSTIC MESSAGES OF SEVERITY W AND ABOVE

ERROR ID L   STMT    MESSAGE DESCRIPTION                                                                  (17)


WARNING DIAGNOSTIC MESSAGES


IEL0916I W   1       ITEM(S) 'ROWPROD','COLPROD','A' MAY BE UNINITIALIZED WHEN USED IN THIS BLOCK.


MESSAGES SUPPRESSED BY THE FLAG OPTION:  4 I.


END OF COMPILER DIAGNOSTIC MESSAGES

COMPILE TIME    0.31 MINS        SPILL FILE:      0 RECORDS, SIZE  4051
```

```
//  EXEC  LNKEDT                                              00410000                 (18)
```

```
JOB  RFP214AB  26/08/76    DOS LINKAGE EDITOR DIAGNOSTIC OF INPUT
                                                                                       (19)
ACTION TAKEN  MAP  REL
LIST   AUTOLINK   IBMBCACA
LIST   AUTOLINK   IBMBCWDH
LIST   AUTOLINK   IBMBOCLA
LIST   AUTOLINK   IBMBPIRA
LIST   AUTOLINK   IBMBERRA
LIST   AUTOLINK   IBMBOCNA
LIST   AUTOLINK   IBMBPGRA
LIST   AUTOLINK   IBMBSDOA
LIST   AUTOLINK   IBMBSIOA
LIST   AUTOLINK   IBMBSLOA
LIST   AUTOLINK   IBMBSPLA
LIST   AUTOLINK   IBMDSTFA
LIST   AUTOLINK   IJJFCBZD
LIST   ENTRY
```

```
26/08/76   PHASE   XFR-AD  LOCORE  HICORE  DSK-AD     ESD TYPE  LABEL     LOADED  REL-FR

       PHASE***  100078  100078  102C9C  0D9 0D 09 CSECT     PLISTART  100078  100078  RELOCATABLE

                                                   CSECT     ******P1  100088  100088                 20
                                                   *  ENTRY  P         10008C

                                                   CSECT     ******P2  100370  100370

                                                   CSECT     PLIMAIN   100568  100568

                                                   CSECT     IJJFCBZD  102918  102918
                                                   *  ENTRY  IJJFCIZD  102918

                                                   CSECT     SYSPINT   100570  100570

                                                   CSECT     IBMBCAC1  100718  100718
                                                      ENTRY  IBMBCACA  100718

                                                   CSECT     IBMBCW01  1009E8  1009E8
                                                      ENTRY  IBMBCWDH  1009E8
                                                   *  ENTRY  IBMBCWZH  1009E8

                                                   CSECT     IBMDOCL1  100BD8  100BD8
                                                      ENTRY  IBMBOCLA  100BD8
                                                      ENTRY  IBMBOCLC  100BDC
                                                      ENTRY  IBMBOCLB  100BDA
                                                      ENTRY  IBMBOCLD  100BDE
                                                      ENTRY  IBMBOCLG  100CA4
                                                      ENTRY  IBMBOCMA  100BD8
                                                   *  ENTRY  IBMBOCMB  100BDA
                                                   *  ENTRY  IBMBOCMC  100BDC
                                                   *  ENTRY  IBMBOCMD  100BDE
                                                   *  ENTRY  IBMBOCMG  100CA4

                                                   CSECT     IBMDPIR1  100CC8  100CC8
                                                      ENTRY  IBMBPIRA  100CF0
                                                   *  ENTRY  IBMBPLRA  100CF0

                                                   CSECT     IBMDERR1  100F00  100F00
                                                      ENTRY  IBMBERRB  100F40
                                                      ENTRY  IBMBERRA  100F00
                                                      ENTRY  IBMBERRC  1014C0

                                                   CSECT     IBMDOCN1  101548  101548
                                                      ENTRY  IBMBOCNA  101548
                                                   *  ENTRY  IBMBRIOC  10156C

                                                   CSECT     IBMDPGR1  101610  101610
                                                      ENTRY  IBMBPGRB  101612
                                                      ENTRY  IBMBPGRC  10162E
                                                      ENTRY  IBMBPGRD  10162A
                                                      ENTRY  IBMBPGRA  101610

                                                   CSECT     IBMDSDO1  101890  101890
                                                      ENTRY  IBMBSDOA  101896
```

```
26/08/76   PHASE  XFR-AD  LOCORE  HICORE  DSK-AD    ESD TYPE  LABEL     LOADED  REL-FR

                                                       ENTRY  IBMBSDOB  101894                    (21)
                                                       ENTRY  IBMBSDOT  101AF2
                                                    *  ENTRY  IBMBSDOC  101892
                                                    *  ENTRY  IBMBSDOD  101890

                                                    CSECT     IBMDSIO1  101D58  101D58
                                                       ENTRY  IBMBSIOA  101D58
                                                       ENTRY  IBMBSIOE  101D60
                                                    *  ENTRY  IBMBSIOB  101D5A
                                                    *  ENTRY  IBMBSIOC  101D5C
                                                    *  ENTRY  IBMBSIOD  101D5E
                                                    *  ENTRY  IBMBSIOT  101EAA

                                                    CSECT     IBMDSLO1  101EC8  101EC8
                                                       ENTRY  IBMBSLOA  101EC8
                                                    *  ENTRY  IBMBSLOB  101ECA

                                                    CSECT     IBMDSPL1  102520  102520
                                                       ENTRY  IBMBSPLA  102520
                                                       ENTRY  IBMBSPLB  102522
                                                       ENTRY  IBMBSPLC  102524

                                                    CSECT     IBMDSTF1  102750  102750
                                                       ENTRY  IBMDSTFA  102764

* UNRESOLVED EXTERNAL REFERENCES                    WXTRN     PLITABS
                                                    WXTRN     PLIFLOW
                                                    WXTRN     PLICOUNT
                                                    WXTRN     IBMBSXCA
                                                    WXTRN     IBMBSXCB
                                                    WXTRN     IBMBSIST
                                                    WXTRN     IBMBCGZA
                                                    WXTRN     IBMBCCQA
                                                    WXTRN     IBMBCHXH
                                                    WXTRN     IBMBCHFY
                                                    WXTRN     IBMBCVDY
                                                    WXTRN     IBMBPGOA
                                                    WXTRN     IBMBJWTA
                                                    WXTRN     IBMBTOCA
                                                    WXTRN     IBMBTOCB
                                                    WXTRN     IBMBILC1
                                                    WXTRN     IBMBPJRC
                                                    WXTRN     PLIXOPT
                                                    WXTRN     PLIXHD
                                                    WXTRN     IBMBERCA
                                                    WXTRN     IBMBKSYA
                                                    WXTRN     IBMBSCPA
                                                    WXTRN     IBMBCBCA

020 UNRESOLVED ADDRESS CONSTANTS
```

```
// EXEC ,SIZE=64K                                                  00420000
```

22

```
PRODUCTS OF ROWS

ROWPROD(1)=      1    ROWPROD(2)=      4    ROWPROD(3)=      9    ROWPROD(4)=     16    ROWPROD(5)=     25
ROWPROD(6)=     36    ROWPROD(7)=     49    ROWPROD(8)=     64    ROWPROD(9)=     81    ROWPROD(10)=   100
ROWPROD(11)=   121    ROWPROD(12)=   144;

PRODUCTS OF COLUMNS

COLPROD(1)=   1600    COLPROD(2)=   1600;
```

23

APPENDIX B. 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 _DOS/VS Access Method Services User's Guide_. 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 functions.

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 system control statements 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 system control statements can be restricted to associating the data set name with the file name and specifying the logical unit. 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 system control statements 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.

A discussion on the physical structure of VSAM data sets is given in Chapter 10; a more thorough discussion is given in _DOS/VS Access Method Services User's Guide_.

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 called from a PL/I user program. In a batch system, the EXEC statement

```
// EXEC IDCAMS,SIZE=nK
```

is used and the commands placed in the file SYSIN. 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(VSER04) CYL(1 1) -
   RECSIZE(20 80) KEYS(10 0))
```

This defines a key sequence data set called BLOGGS on the volume VSER04. 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. 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:

1. Definition with the DEFINE command.

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.

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 CLUSTER 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 can 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 explained in the DOS/VS Access Method Services User's Guide. 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. A simplified form of the command is:

Syntax

```
DEFINE CLUSTER (NAME(data-set-name)
     CYLINDERS(primary
                [ secondary]) |
     RECORDS(primary
                [ secondary]) |
     TRACKS(primary
                [ secondary])]
     [FILE(dname)]
     [FREESPACE(cipercent
                [ capercent])]
     [INDEXED|NONINDEXED|
                NUMBERED]
     [KEYS(length offset)]
     [RECORDSIZE(average maximum)]
     [REUSE|NOREUSE]
     [SHAREOPTIONS(crosspartition
                [ crosssystem])]
     [VOLUMES(volser[ volser...])]
     [password-options]
     [other-options]
   [DATA(option-list)]
   [INDEX(option-list)]
   [CATALOG(catname[/password]
                [ dname])]
```

Items in uppercase (capital letters) must be coded as shown. Items in lowercase must be replaced by the information you require; hyphens indicate that the items replaced should not contain blanks. Alternatives are separated by the vertical stroke, |. Underscored items indicate default options. Items enclosed in square brackets are optional. 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; 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.

- 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.

Most DEFINE CLUSTER commands should specify 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, refer to the DOS/VS Access Method Services User's Guide.

NAME(data set name**)**
specifies the name of the data set. NAME must be specified for the cluster. If no name is specified for a data set, a name is generated and listed for you.

The name may contain from 1 through 44 alphameric characters, national characters (@, #, and $), and two special characters (the hyphen and the 12-0 overpunch). Names containing more than eight characters must be segmented by periods; one to eight characters may be specified between periods. The first character of any name or any name segment must be either an alphabetic or a national character.

CYLINDERS(primary[secondary]**) |**
RECORDS(primary[secondary]**) |**
TRACKS(primary[secondary]**)**
specifies the space that is to be reserved for your data set, in either cylinders, records, or tracks. 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.

FILE(dname**)**
specifies the filename of the DLBL job control statement that, together with an EXTENT statement, identifies the logical units and volumes to be used for space allocation.

FREESPACE(cipercent[capercent]**)**
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.

cipercent
is the percentage of each control interval that is to be left empty.

capercent
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 DOS/VS Access Method Services User's Guide).

The default is FREESPACE(0 0).

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

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.

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.

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.

SHAREOPTIONS(n[m]**)**
is described below under "Sharing a Data Set between Jobs" on page 304.

VOLUMES(volser1 [volsern]**)**
specifies the volume or volumes on which your data set is to reside. A volume serial number, volser, may contain one to six alphameric, national (@, #, and $), and special characters (commas, blanks, semicolons, parentheses, slashes, asterisks, periods, quotation marks, ampersands, plus signs, hyphens, and equal signs). Single quotation marks within a volume serial number must be coded as two single quotation marks. A volume serial number must be enclosed in single quotation marks if it contains a special character.

For consistency with DOS/VS job control statements, only alphameric characters should be used.

password-options
specify the password or passwords for your data set. Password levels differ for various degrees of security. These levels are (from low to high):

READPW(password**)**
    gives read-only access.

UPDATEPW(password**)**
    gives access to alter contents.

CONTROLPW(password**)**
    is irrelevant to PL/I users.

MASTERPW(password**)**
    gives complete access to data set.

password is a 1 to 8 EBCDIC character password.

If only a low-level password such as READPW is specified, the read password is propagated upward so that it also becomes the other passwords. If only a high-level password is specified, lower level passwords will not be required.

other-options
: Numerous other options can be specified that control the physical structure, data integrity, and protection of VSAM data sets. See the DOS/VS Access Method Services User's Guide.

DATA(option-list)
: specifies attributes of the data set of the cluster. For information on these attributes (option-list), refer to the DOS/VS Access Method Services User's Guide.

INDEX(option-list)
: specifies, for a key-sequenced file, attributes of the index data set of the cluster. For information on these attributes (option-list), refer to the DOS/VS Access Method Services User's Guide.

CATALOG(catname[/password][dname])
: identifies the catalog in which the cluster is to be defined.

  catname
  : specifies the name of the catalog.

  password
  : specifies the update or higher-level password if the catalog is password protected.

  dname
  : specifies the filename of the DLBL job control statement that identifies the catalog.

An example of the use of the DEFINE CLUSTER command is:

```
  DEFINE CLUSTER -
     (NAME(EXAMPLE.ONE) -
      READPW(ONEPSWD) -
      VOL(VSER04) -
      RECORDSIZE(400,475) -
      KEYS(12 4) -
      FREESPACE(40,40) -
      TRACKS(10 5)) -
     CATALOG(AMASTCAT/MCATUPPW)
```

This example defines a key-sequenced data set into the master catalog. The key is 12 bytes long and starts at offset 4 (the fifth character). Forty percent of each control interval, and forty percent of the control intervals in each control area, will be kept empty for new records. The primary space allocation is 10 tracks and the secondary allocation will be in increments of 5 tracks. The catalog in which the data set is to be defined is called AMASTCAT and the password is MCATUPPW.

Complete examples of PL/I statements, system control statements, and Access Method Services commands are given at the end of Chapter 10.

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. For example:

```
  // JOB  ....
  // EXEC IDCAMS,SIZE=64K
       DEFINE CLUSTER (NAME(FRED) -
               VOLUMES(VSER05) -
               TRACKS(10 5) -
               RECORDSIZE(80 100) -
               NONINDEXED -
           CATALOG(MASTCAT)
  /*
```

SHARING VSAM DATA SETS

The extent to which VSAM data sets can be shared depends upon the SHAREOPTIONS specified in the DEFINE CLUSTER command when the data set is defined. A description of sharing between jobs and sharing within a job follows. For detailed information, refer to DOS/VS Access Method Services User's Guide.

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, where:

n specifies cross-partition sharing and has the following meanings:

1

specifies that any number of users can share the component or cluster being defined if the file is opened for input (read operations) only; however, once a file has been opened for input, it cannot be opened for output. Conversely, once the file has been opened for output (write operations), no other user can open it for either input or output until that operation is finished.

2

specifies that any number of users can use the component or cluster for input (read operations) even if one user is using it for output (write operations). No more than one user can open the file for output at a time.

3

specifies that any number of users can share the component or cluster for both read and write operations. VSAM does not monitor accesses to ensure data integrity.

4

specifies that any number of users can share the component or cluster for both read and write operations. VSAM ensures write integrity for ACBs opened from different tasks or partitions. If the user issues only GET updates throughout the system, read integrity will also result from this option. The DOS/VS trackhold facility is used to ensure the appropriate integrity.

and **m** specifies cross-system sharing. Cross-system sharing has no effect on DOS/VS itself. However, if it were specified and the pertinent cluster were then imported into an OS/VS system, the cross-system subparameter would be effective for values 3 or 4 only. The meanings for these values are the same as their meanings for cross-partition sharing.

When a data set is opened, VSAM checks to see if it is being shared. If it is, VSAM checks to see whether the type of sharing requested is allowed in the SHAREOPTIONS. If it is not allowed, the file is not opened and the UNDEFINEDFILE condition is raised.

Sharing within a Job

Data sets can be shared within a job by having a number of DLBL 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 CLUSTER (data-set-name
                [/password)]
      [CATALOG(catname[/password]
              [ dname])]
      [other-options]
```

data-set-name
: is the name of the data set that you want to delete.

password
: is the master password for the data set.

CATALOG[catname**[/**password**][** dname**])**
: specifies the name of the catalog that defines the data set to be deleted.

  catname
  : identifies the catalog.

  password
  : specifies the master password of the catalog. If data sets to be deleted are password protected and the catalog is also password protected, a password must be supplied either through CATALOG or with the name of each data set to be deleted.

  dname
  : specifies the filename of the DLBL job control statement which identifies the catalog that defines the data sets to be deleted. The dname is required if the desired catalog is not either the master or job catalog.

other-options
: specifies other facilities of the DELETE command. These are described in the DOS/VS Access Method Services User's Guide.

An example of deleting a data set in a batch programming environment is:

```
// JOB .....
// EXEC IDCAMS,SIZE=64K
   DELETE FRED CATALOG(MASTCAT)
/*
```

This deletes the data set FRED defined in the example of the DEFINE CLUSTER command shown earlier in this section.

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 nonunique key. For a unique key alternate index, each record must have a different key. For a nonunique 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 nonunique key alternate index because more than one person would be in each department. An example of alternate indexes applied to a family tree is given in Figure 59 on page 170.

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 61 on page 179. 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 nonunique key alternate index path, the record accessed is determined by the key and the sequence. The key can be used to establish positioning so that sequential access may follow. The use of the key accesses the first record with that key. When the data set is read backward, 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 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 automatically 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 103 on page 308.

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 nonunique
- Whether the index is to be password protected
- Some of the performance aspects of using alternate indexes.

The type of the base cluster and the use of unique or nonunique 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 60 on page 173 and Figure 61 on page 179, respectively, show 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 to access a data set via an alternate index path in another.

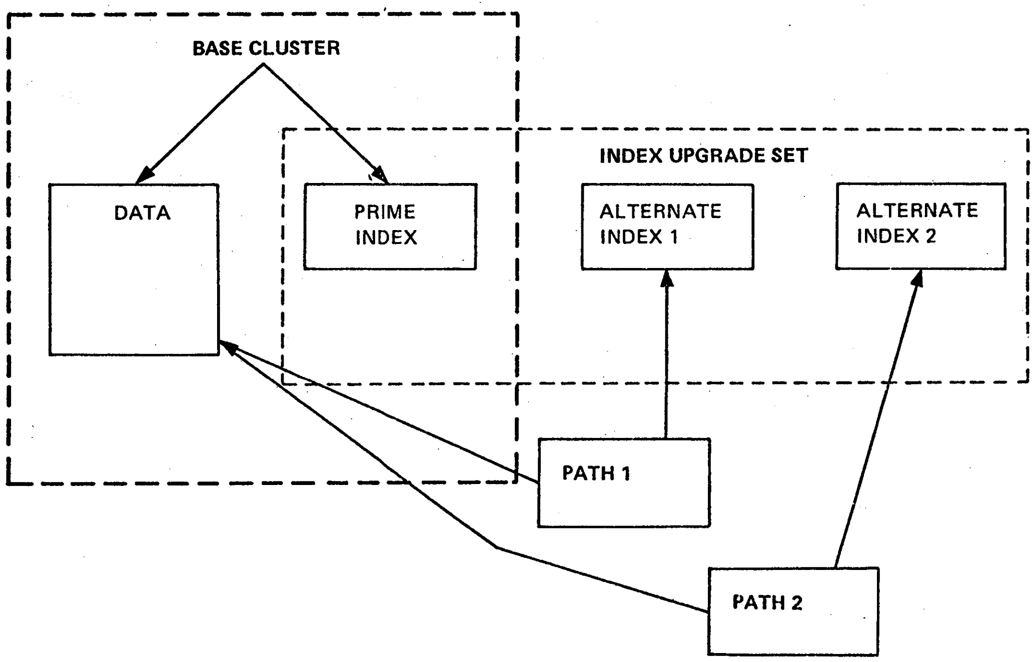
Passwords

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. The use of alternate indexes, however, causes the access of a record to be less direct and therefore not as fast. When a data set with a number of indexes is opened, the indexes are by default opened at the same time as the data set to allow for possible upgrade.

If you are using the data set for read-only processing, however, you do not need to upgrade the alternate indexes, and you can improve performance by defining a path with the NOUPDATE attribute. (The method of defining such a path is described in the MVS/Extended Architecture VSAM Administration Guide.) NOUPDATE prevents the upgrade of alternate indexes for the data set. For this reason, you should not alter the data set using a path defined with the NOUPDATE attribute.



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 103. Base Cluster, Alternate Indexes, and Paths

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 nonunique
- If the keys will be nonunique, 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 can 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 job step is given at the end of this section.

DEFINE ALTERNATEINDEX Command

A simplified form of the DEFINE ALTERNATEINDEX command is:

Syntax

```
DEFINE ALTERNATEINDEX
    (NAME(indexname)
     RELATE(data-set-name)
            [/password])
    [CYLINDERS(primary
            [ secondary]) |
     RECORDS(primary
            [ secondary]) |
     TRACKS(primary
            [ secondary])]
    [FILE(dname)]
    [FREESPACE(cipercent
            [ capercent])]
    [KEYS(length offset)]
    [RECORDSIZE(average maximum)]
    [REUSE|NOREUSE]
    [SHAREOPTIONS(crosspartition
            [ crosssystem])]
    [UNIQUEKEY|NONUNIQUEKEY]
    [UPGRADE|NOUPGRADE]
    [VOLUMES(volser1
            [ volsern. . .])]
    [password-options]
    [other-options]
  CATALOG(catname[/password][ dname])]
```

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" earlier in this chapter.

NAME(<u>indexname</u>**)**
specifies the name of the alternate index. The name may contain from 1 through 44 alphameric characters, national characters (@, #, and $), and special characters (hyphen and 12-0 overpunch). Names containing more than 8 characters must be segmented by periods; one to eight characters may be specified between periods. The first

character of any name or name segment must be either an alphabetic or a national character.

RELATE(data-set-name**[**/password**])**
specifies the base data set with which the alternate index will be associated.

KEYS(length offset**)**
specifies the position of the alternate index key in the record. They may be anywhere within the record.

RECORDSIZE(average maximum**)**
specifies the size of the record in the alternate index. If the path is nonunique, each index record will have to refer to many data records. Consequently, if the key is nonunique, the maximum should be a large figure. The default values are large; see the DOS/VS Access Method Services User's Guide.

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 automatically updated whenever the base data set is altered (using this index or any other index). If NOUPGRADE is specified, the index is not automatically altered with the data set.

password-options
CONTROLPW, MASTERPW, READPW, and UPDATEPW are the password options of the alternate index. See "DEFINE CLUSTER Command" on page 300 in this chapter for details.

other-options
are described in the DOS/VS Access Method Services User's Guide.

CATALOG(catname**[**/password**][** dname**])**
specifies the catalog in which the alternate index will be defined. It must be the same as the catalog of the base data set.

See "DEFINE CLUSTER Command" on page 300 for details.

An example of the DEFINE ALTERNATEINDEX command is:

```
  DEFINE ALTERNATEINDEX -
      (NAME(ALPHINDX) -
      VOLUMES(VSER04) -
      KEYS(10 0) -
      NONUNIQUEKEY -
      RELATE(PERSNOS) -
      RECORDSIZE(20 2000)) -
  CATALOG(AMASTCAT/MCATUPPW)
```

This defines an alternate index called ALPHINDX on the data set called PERSNOS. The keys are nonunique and are in the first 10 bytes of the record. It is cataloged in the catalog called AMASTCAT with the master password MCATUPPW.

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. DLBL 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(dname1[/read-password])
 OUTFILE(dname2[/update-password])
[CATALOG(catname[/update-password])]
```

where dname1 is the name of the base data set and dname2 is the name of the alternate index. For example:

```
BLDINDEX INFILE(BASE) -
    OUTFILE(ALTIND) -
    CATALOG(AMASTCAT/MCATUPPW)
```

The DLBL and EXTENT statements take the following forms:

Base Cluster:

```
// DLBL   BASE,'ALPHIND',,VSAM
// EXTENT SYS007,VSER04
```

where BASE is the INFILE name and ALPHIND is the data set name.

Alternate Index:

```
// DLBL   ALTIND,'ALPHIND',,VSAM
// EXTENT SYS007,VSER04
```

where ALTIND is the OUTFILE name and ALPHIND is the data set name.

Sort Workfiles:

```
// DLBL   IDCUT1,'SORT.WORK.ONE',,VSAM
// EXTENT SYS008,VSER05
// DLBL   IDCUT2,'SORT.WORK.TWO',,VSAM
// EXTENT SYS008,VSER05
```

A combined example showing the commands and job control statements required to create an alternate index is given in Figure 104 on page 313.

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[/password])
 [password-options]
 [other-options]
[CATALOG(catname[/password][ dname])]
```

The DEFINE PATH command and its options are described in detail in the DOS/VS Access Method Services User's Guide.

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 104 on page 313 shows the use of Access Method Services in a batch system. 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 that 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 VSER4.

The example is commented to aid understanding. Access Method Services comments are delimited by /* and */. System control comments are one line in length and start with //*. These are the allowed forms for comments.

DELETING AN ALTERNATE INDEX

Alternate indexes and alternate index paths are automatically 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.

```
// JOB    .....
* DLBL and EXTENT statements, for BLDINDEX command follow
* first the alternate index
// DLBL   ALTIND,'ALPHIND',,VSAM
// EXTENT SYS007,VSER4
* then the base data set
// DLBL   BASEDS,'PERSNOS',,VSAM
// EXTENT SYS007,VSER4
* the DLBL and EXTENT statements for BLDINDEX sort files follow
// DLBL   IDCUT1,'SORT.WORK.ONE',,VSAM
// EXTENT SYS006,VSER5
// DLBL   IDCUT2,'SORT.WORK.TWO',,VSAM
// EXTENT SYS006,SER5
// EXEC  IDCAMS,SIZE=64K
     DEFINE ALTERNATEINDEX -
         (NAME(ALPHIND)          /* data set name of alternate index */ -
          VOLUMES(VSER4)         /* 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 */

     BLDINDEX -                  /* this command loads the data into the alternate
                                    index created in the previous command */ -
         INFILE(BASEDS)          /* name of base data set */ -
         OUTFILE(ALTIND)         /* 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 data
                                    set name in DLBL statement PL/I program */ -
              PATHENTRY(ALPHIND)) /* name of alternate index */ -
            CATALOG(NMCAT)
/*
```

In this example, there are five names involved:

1. The data set name (or file identifier) of the base data set—PERSNOS. Used in the RELATE operand of the DEFINE ALTERNATEINDEX command, and as the filename in the DLBL statement for the INFILE of the BLDINDEX command.
2. The filename of the base data set—BASEDS. Used in the INFILE operand of the BLDINDEX command and as the name in the DLBL statement for the INFILE.
3. The data set name of the alternate index—ALPHIND. Given in the NAME operand of the DEFINE ALTERNATEINDEX command, and used as the data set name (file identifier) in the DLBL statement for the BLDINDEX OUTFILE, and in the PATHENTRY operand of the DEFINE PATH command.
4. The filename of the alternate index—ALTIND. Used in the OUTFILE operand of the BLDINDEX command and as the filename in the DLBL 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 data set name (file identifier) when the base data set is accessed through the alternate index paths.

Figure 104. Commands Required to Create an Alternate Index Path

APPENDIX C. COMPATIBILITY WITH THE DOS PL/I D COMPILER

Some features of the DOS PL/I Optimizing Compiler implementation are incompatible with the language implemented by version 4 of the PL/I D Compiler. The most significant incompatibilities are listed below. Except where stated, the description given is of the optimizing compiler implementation. Programs written for version 4 of the PL/I D Compiler that use any of these features should be reviewed before compiling them with the optimizing compiler, to ensure that they will return the same results.

A number of the differences given here are also given in the general information manual for this compiler. The general information manual also contains some of the implementation limitations and restrictions restrictions of this compiler. The language reference manual for this compiler gives full details of the implementation of each language feature.

Alignment of Strings

The UNALIGNED attribute is implemented in full by the optimizing compiler. The D compiler does not implement UNALIGNED for bit strings and forces the alignment of all character strings.

Because the default for alignment is UNALIGNED, conversion problems that may result can be overcome by including the statement:

```
DEFAULT RANGE(*) ALIGNED;
```

This will cause all strings in the converted program to remain aligned.

Assembler Language Interface

Assembler language subroutines used with PL/I programs written for the D compiler may have to be modified when the programs are recompiled by the optimizing compiler. Whereas a D compiler program passes arguments to an assembler routine as the addresses of data items, in many cases, optimizing compiler programs will pass the address of locator/descriptors for the data items.

Built-in Function without Arguments

Built-in functions without arguments, such as TIME and DATE, must be declared explicitly with the attribute BUILTIN.

Expressions in DO Statements

Expressions in DO statements are evaluated by the D compiler in the order "expression2" followed by "expression3", irrespective of the order of appearance in the DO statement. For example:

```
DO I = J TO K BY L;

DO I = J BY L TO K;
```

In both statements, "expression2" is represented by K and "expression3" by L. The D compiler always evaluates expression2 first, whereas the order in which the optimizing compiler evaluates each expression is undefined.

SYSIN and SYSPRINT

Although the names SYSIN and SYSPRINT have no special meaning for the D compiler, they do for the optimizing compiler. PL/I programs can contain stream-oriented (GET or PUT) data transmission statements which do not specify a file name. The D compiler treats such statements as referring to the symbolic devices SYSIPT and SYSLST; the optimizing compiler makes the assumption that such input statements refer to SYSIN, and output statements to SYSPRINT.

E- and F-format Items

Zero before decimal point: When F-format fractional values or E-format zero mantissa values are transmitted, the optimizing compiler inserts a leading zero before the decimal point. The D compiler does not put the zero before the point. For example:

```
D compiler           -.500
Optimizing compiler  -0.500
```

Buffered Data Sets

If errors are detected in a LOCATE statement, space will not be allocated in the buffer and the pointer will not be set.

REGIONAL Data Sets

REGIONAL data sets for programs written for the D compiler, are, when created, preformatted by a utility program. This program is executed as a separated job step prior to execution of the PL/I program in which the output file is opened to create the data set. Subsequent use can be made of this data set through an OUTPUT file without formatting it again.

REGIONAL data sets created for programs compiled by the optimizing compiler are preformatted by a PL/I library subroutine when the output file is opened. Thus an output file cannot be opened to process a regional data set without destroying all the records contained in it. If records are to be added to the regional data set, an UPDATE file must be used.

Preformatting, including the preformatting of secondary extents, is performed as follows:

1. A REGIONAL(1) data set with the the attributes DIRECT and OUTPUT, is preformatted with dummy records when the file is opened. (A dummy record is a record whose first byte is set to X'FF' and whose remaining bytes are undefined.)

2. A REGIONAL(1) data set with the attributes SEQUENTIAL and OUTPUT has all tracks cleared when the file is opened. Dummy records are written into those regions that do not receive a data record during processing.

3. A REGIONAL(3) data set with the attributes SEQUENTIAL and OUTPUT, or DIRECT and OUTPUT has all tracks cleared when the file is opened.

Because dummy records can be retrieved from a REGIONAL(1) data set by a READ statement, the programmer must ensure that dummy records are recognized by the program.

Halfword Binary Numbers

Fixed-point binary numbers with a precision of (15) or less are held in main storage as halfword binary numbers by programs compiled by the optimizing compiler. Fixed binary numbers with a precision greater than (15) are held as fullword binary numbers. All fixed-point binary numbers in programs compiled by the D compiler are held as fullword binary numbers.

D compiler programs to be recompiled should be checked for occurrences of FIXED BINARY variables which have precisions of (15) or less (thus including those with default precision), since they might occur in record-oriented transmission and cause differences in record lengths and in the alignment of records in locate-mode buffers. A similar problem could occur for programs that process data sets created by D compiler object programs. Bit-string values returned by the UNSPEC built-in function when used with halfword binary numbers as arguments are 16 bits in length. The DEFAULT statement may be used to ensure that all undeclared fixed binary variables have the maximum precision (31,0).

Labels on DECLARE Statements

The D compiler ignores any labels prefixed to DECLARE statements. The optimizing compiler recognized such labels and treats branches to such labels as branches to null statements. An incompatibility can occur if in a recompiled D compiler program such a label has the same identifier as a variable or is used as a label prefix to another statement.

ONSYSLOG Option

The optimizing compiler does not support the use of the ONSYSLOG option, whereby all output resulting from actions derived from on-conditions is printed on the system log.

DYNDUMP

The optimizing compiler does not permit use of the DYNDUMP, IJKTRON, IJKTROF, and IJKEXHC subroutines.

DISPLAY Statement and REPLY Option

The optimizing compiler permits strings up to 126 bytes in length for both the DISPLAY statement and the REPLY option. The D compiler permits strings up to 80 bytes for the DISPLAY statement, and up to 256 bytes for the REPLY option.

INDEX Built-in Function

The INDEX built-in function can be used with a binary arithmetic argument that requires conversion to character string form before the built-in function can be executed. This occurs wherever the other argument is either a decimal arithmetic value or a character string. For example:

```
INDEX(A,I)

INDEX(I,'B')
```

In both cases I is a binary arithmetic variable. A is a decimal arithmetic variable. In the first case both A and I are converted to character form, in the second case only I is converted.

An incompatibility exists between the methods and the results of conversion from binary arithmetic to character form for this built-in function. The D compiler converts a binary arithmetic argument to an intermediate bit-string form which is then converted to character form consisting only of ones and zeros. The optimizing compiler converts the argument to an intermediate decimal arithmetic form, which is then converted to character form consisting of all numeric characters.

PRECISION Built-in Function

The PRECISION built-in function is implemented differently by the D and optimizing compilers. For the D compiler, if the first argument is FIXED, and the third argument is omitted, the third argument is assumed to be zero, and the compiler will issue an informatory message.

Redundant Expression Elimination

The optimization processor eliminating redundant expressions could give rise to an incompatibility for D compiler programs that are recompiled by the optimizing compiler. If a program contains an expression, such as IF (A=D)|(C=D) THEN...such that the condition (A=D) is satisfied, the expression (C=D) is ignored. However, (C=D) might contain a function which, if not evaluated, could give rise to error.

SUM and PROD Built-in Functions

For the optimizing compiler, the SUM and PROD built-in functions accept arguments that can be arrays of either fixed-point or floating-point elements. The value returned is in the same scale as the argument given, except for the PROD built-in function used with fractional fixed-point arguments, where the value returned is in floating-point scale. Note that string arguments are converted to fixed-point arithmetic form, and that the result is returned in this form.

For the D compiler, the arguments of these built-in functions are, if necessary, converted to floating-point scale. The returned value always has floating-point scale.

Attributes of File Parameters

For the D compiler, a file parameter can be declared with other attributes in addition to the FILE attribute. For the optimizing compiler, a file parameter can only be declared with the FILE attribute; all other attributes are inherited from the argument. If additional attributes are given, the compiler will issue an informatory message, and ignore them.

Defining of Pictures

Simple defining of pictures will be diagnosed by the optimizing compiler as an error if the defined element is a picture that does not exactly match the base element. The D compiler requires only that the base element should be a picture or a character string.

Sterling Pictures

Sterling data is not supported by the optimizing compiler. A picture including any of the following characters is invalid:

G, M, H, P, 6, 7, 8

Source Program Errors

The D compiler does not detect all the errors in a source program that can be detected by the optimizing compiler. Errors that the D compiler does not detect include the transfer of control into an iterative do-group, comparison of structures, and incorrect overlay defining.

Programs which contain these errors and compile successfully with the D compiler will not compile successfully with the optimizing compiler.

RETURNS Keyword in PROCEDURE and ENTRY Statements

PROCEDURE and ENTRY statements for function procedures that specify the attributes of the value returned by the procedure must, for the optimizing compiler, have such attributes contained in a parenthesized list preceded by the keyword RETURNS. For example, the following statement is valid for the D compiler, but not for the optimizing compiler:

```
  X: PROCEDURE (Y,Z) FLOAT BINARY;
```

For the optimizing compiler, this statement should be written as follows:

```
  X: PROCEDURE (Y,Z) RETURNS (FLOAT BINARY);
```

Entry Names as Arguments

The D compiler assumes an entry name argument in parentheses and without arguments of its own to be a function reference. For example, in the expression X((Y)), the function Y is invoked and the value it returns is used as the argument to procedure X. The optimizing compiler assumes that the entry name itself is to be passed as an argument. It creates an entry variable with the value of the entry constant argument, and passes this as a dummy argument to the invoked procedure. Function references such as this in programs written for the D compiler should be modified to contain a null argument list in order to invoke the function. For example, the expression given above should be written as X(Y()).

ENDFILE Condition

For the D compiler, after the ENDFILE condition has been raised once, a subsequent execution of a GET or READ statement will not raise ENDFILE again, but will read past the file delimiter. For the optimizing compiler, subsequent GET or READ statements will raise ENDFILE again.

MEDIUM Option

The optimizing compiler allows specification of MEDIUM without a physical device type. This is used to request the generation of device independent tables. The D compiler generates device independent tables no matter what device type is specified provided all other necessary restrictions are observed.

SIZE Condition

For the D compiler, if the SIZE condition is raised during E- and F-format output, asterisks are transmitted. For the optimizing compiler, the results is undefined.

INITIAL Attribute and Statement Length

The optimizing compiler has a restriction that any statement must fit into a work area used by the compiler. The maximum size of the work area varies with the amount of main storage available to the compiler. The limitations on the length of statements are as follows:

Space Available	**Maximum Statement Length**
50K - 55K bytes	1012 characters
55K - 69K bytes	1600 characters
69K - 75K bytes	3400 characters

Note: The values 55K and 69K for the changeover points are only approximate.

A maximum statement length of 4000 characters can be obtained only if the space available is greater than 75K and if the IBM 3330, 3340, 3350, 3375, 3380, or a fixed block device is used for the spill file.

The DECLARE statement is an exception in that it can be regarded as a sequence of separate DECLARE statement separated by those commas which are not contained in parentheses.

For example,

```
  DCL 1 A,
        2 B (10,10) INITIAL (1,2,3,...),
        2 C (10,100) INITIAL ((1000) 0 ),
        2 (D,E) CHARACTER (20) VARYING,
                     .
                     .
                     .
```

In this example, each line is terminated by a comma that is not contained within parentheses. Consequently, the compiler can treat each line as a separate DECLARE statement insofar as the use of its work area is concerned.

The compiler will also permit a DECLARE statement to exceed the size of the work area if it contains an INITIAL attribute that specifies a simple list of items and that is not within any brackets. For example:

```
   . . .  INITIAL (item,item,item,...)...
```

In this case, for the purpose of its work area, the compiler will split the declaration at one or more of the commas in the list. Each item created by the split may contain initial values that, when expanded to eliminate any repetition and iteration factors, do not exceed the maximum statement length.

The above paragraph also applies to the use of the INITIAL attribute in a DEFAULT statement.

The D compiler makes a special case of INITIAL clauses, and it is possible, therefore, that a program which can be compiled by the D compiler may exceed the limit for the optimizing compiler.

If the problem is encountered, it may be solved by one of the following techniques:

- Increase the main storage available to the compiler (unless it already exceeds 69K).
- Simplify the DECLARE statement so that the compiler can split the statement.
- 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 the work area:

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

Use of the DEFINED Attribute

The PL/I D compiler does not implement simple defining. Instead, the D compiler treats instances of simple defining as string overlay defining. Consequently, programs written for the D compiler that contain instances of simple defining will produce different results when recompiled by the optimizing compiler. For example:

```
  DCL A(10) CHAR(8),
     B(10) CHAR(4) DEFINED(A);
```

In this example, the D compiler overlays the array B on the first 40 characters (the first five elements) of array A. Consequently, a reference to B(2) is equivalent to a reference to SUBSTR(A(1),5,4). The optimizing compiler will simply overlay each element of the array B on the corresponding element of array A. Consequently, a reference to B(2) is then equivalent to a reference to SUBSTR(A(2),1,4).

Another example follows:

```
  DCL  1  S,
       2  A CHAR(8)
       2  B CHAR(72),
       1  S1 DEFINED S,
       2  A1CHAR(4),
       2  B1 CHAR(4);
```

In this example, a reference to element B1 is interpreted by the D compiler as a reference to SUBSTR(A,5,4) and by the optimizing compiler as a reference to SUBSTR(B,1,4).

To achieve compatibility in such cases, the attribute POS(1) must be included in the declarations of the defined item before the program is recompiled by the optimizing compiler.

APPENDIX D. PL/I-CICS SYSTEM INFORMATION

Each of the two PL/I-CICS interfaces supplies a module (DFHSAP) to be loaded as a part of the CICS/VS nucleus and a module (DFHPL1I) 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 DFHPL1I, and vice versa. A single execution of CICS/VS can load only one DFHSAP; therefore, all PL/I-CICS/VS transaction programs in a single execution of CICS/VS (for instance, within a single partition) must use either the CICS/VS-supplied interface or the PL/I-supplied interface, but no intermixing of the two is permitted. Under the current version of CICS/VS, you might use one PL/I interface in one partition and the other PL/I interface in another, even though the CICS/VS systems are not being executed independently of each other. If mixing should inadvertently occur, the results will be as shown in Figure 105 on page 322.

The system programmer should ensure that the proper DFHSAP module is loaded with the CICS nucleus, and that the proper DFHPL1I module is link-edited into transaction programs. (See DOS PL/I Optimizing Compiler: Installation for details.) It is sometimes helpful, however, to know which version of these modules is present. This can be determined as follows:

- For DFHSAP, look at its link-edit listing (or a listing produced by RSERV) to see if it contains external names beginning with IBMF or IBMH. If it does, it is the PL/I-supplied DFHSAP. If no such names are found, it is the CICS-supplied DFHSAP.
- For DFHPL1I, look at its link-edit listing (or a listing produced by RSERV) to see what addresses are represented by entry-point names DFHPL1I, DFHPL1N, and DFHPL1C. If each points to a different location in DFHPL1I, it is the PL/I-supplied DFHPL1I. If they all point to the same location in DFHPL1I, it is the CICS-supplied DFHPL1I.

CICS/VS-PL/I INTERFACE COMPONENTS

PL/I supplies an interface module called DFHPL1I and a module called DFHSAP. DFHSAP is a part of the PL/I product, not of CICS/VS, but it is loaded during CICS/VS initialization to become part of the CICS nucleus. It is supplied as part of the PL/I Transient Library.

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 (even for the CICS/DOS/VS environment).

Certain other functions, normally required only in a debugging environment, are implemented by loading PL/I transients into CICS/VS storage via a DFHPC TYPE=LOAD macro. 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 modules (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/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 they are being executed in the CICS environment by testing a bit, TTKK, in the PL/I TCA. 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

		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 will result.	Supported - PL/I function and restrictions as documented in this publication.

Figure 105. DFHPL1I Link-Edited into Transaction

other library modules have sections of code for the CICS environment that test this bit and use the CICS appendage.

The CICS appendage is equivalent to the following PL/I structure:

```
DCL 1 IBMDZCIC ALIGNED,
      2  TCTCA PTR,
         /* ADDR CICS TCA */
      2  TCCSA PTR,
         /* ADDR CICS CSA */
      2  TCSTV PTR,
         /* ADDR SYSPRINT XMTR */
      2  TCTMS PTR,
         /* ADDR MSGS OUTPUT BOOTSTRAP */
      2  TCTCR PTR,
         /* ADDR COUNT OUTPUT MODULE */
      2  TCRHD CHAR (8),
         /* TRANSID||TERMID */
      2  TCMSP CHAR (1),
         /* PL/I PROGRAM MASK */
      2  TCMSC CHAR (1);
        /* CICS PROGRAM MASK */
```

The module addresses in this appendage are 0 if the particular module has not been loaded for this CICS/VS transaction program. This appendage makes the CICS environment addressable from within the PL/I environment.

CICS/VS-PL/I APPLICATION PROGRAM INTERFACE

The link-edited CICS/VS interface module (DFHPL1I) replaces the batch-mode PLISTART CSECT, and contains what amounts to a CICS/VS-tailored version of the PLISTART parameter list.

Execution of your PL/I transaction program commences when DFHPCP calls DFHPL1I at its entry point DFHPL1N. DFHPL1N immediately calls PL/I initialization in DFHSAP, passing it the following addresses:

PLIMAIN	Address of MAIN procedure.
PLIFLOW	Flow trace initialization module (if FLOW option).
PLICOUNT	Count initialization module (if COUNT option).
PLIXHD	Your heading for COUNT and REPORT output.
PLITCIC	Entry to CICS/VS HLPI.

IBMBPSRA	Shared library transfer vector.
IBMBPOPT	Compiler-parsed PLIXOPT options.
IBMBERCA	CHECK module.

It also passes the length of pseudo-register vector (PRV), always zero for DOS.

Any of the above parameter addresses, except PLIMAIN, can be zero if the related entity or option does not exist for the particular program being initialized.

To pass the above parameter list to PL/I initialization in DFHSAP, DFHPL1I contains the following functional entry points, in addition to the code at entry point DFHPLIN:

DFHPL1I	Bootstrap to CICS services for the macro interface.
DFHPL1C	Entry point to return CSA address to caller.
IBMBOCLA	Entry points to branch through DFHSAP.
IBMBOCLB	OPEN/CLOSE code in STREAM PRINT transmitter IBMBOCLC for SYSPRINT.

CICS/VS-PL/I NUCLEUS MODULE DFHSAP

The PL/I interface module in the CICS/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/VS environment. These modules have names that begin IBMF instead of IBMD. Any DOS-only versions have names that begin with IBMH.

In general, PL/I library module names begin with IBM, have a fourth letter to specify the relevant environment (for example, B for basic, D for DOS only, F for CICS/VS, H for CICS/DOS/VS only, and so on), and a last (eighth) letter to differentiate among entry points (A, B, and so on). Thus, DOS PL/I initialization is done by a module with entry names IBMDPIRA, IBMDPIRB, and IBMDPIRC. Ignoring the different entry points, it is sometimes called IBMDPIR. Ignoring environment, it can be functionally identified as PIR.

Keeping these naming conventions in mind, DFHSAP contains:

IBMFPCC
Bootstrap to library modules in DFHSAP.

IBMBOCLA, B, C
IBMFSTVA, B, C
Bootstrap code to invoke (via CICS LOAD the first time) the STREAM OUTPUT PRINT transmitter for SYSPRINT (plus OPEN/CLOSE).

IBMHPIR
Initialization/termination code, containing all relevant functions of PIR, PII and PIT.

IBMFPGR
Storage management without REPORT option.

IBMFERR
PL/I error handler.

IBMDPSR
Bootstrap module for PL/I CICS/DOS/VS shared library. If it is present, the DOS shared library modules themselves follow immediately as part of DFHSAP.

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 usually 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 loaded into CICS storage by a CICS DFHPC TYPE=LOAD macro. CICS/VS regards them as ordinary transaction programs. 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/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, ESNA, FOCA, ETXA
Messages modules.

IBMFKMPA, 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. They are:

IBMDSIO
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.

IBMDKDM
The resident interface to PLIDUMP loads the first PLIDUMP transient, as appropriate, based on whether the program is being executed within the CICS/VS environment or the normal DOS environment.

APPENDIX E. THE VTOC DISPLAY UTILITY

The VTOC (Volume Table of Contents) display utility program displays the contents of the volume table of contents of a disk storage volume mounted on a 2311, 2314, 3330, or 3340 disk storage drive. This information is essential for controlling the organization of a storage volume. The program can produce its output on a printer, a magnetic tape, or a disk storage volume.

The data sets contained on a disk storage volume are identified by entries in the volume table of contents of a disk storage volume mounted on a 2311, 2314, 3330, or 3340 disk storage drive. This information is essential for controlling the organization of a storage volume. The program can produce its output on a printer, a magnetic tape, or a disk storage volume.

The data sets contained on a disk storage volume are identified by entries in the volume table of contents which serve as data set header labels. The labels in the VTOC are listed in the order of appearance. The listing contains the format of the data set label and the major fields within each label. The VOL1 label, which contains the address of the VTOC, the serial number, and other volume identification, will also be displayed. For more information about the organization of the VTOC and the VOL1 label, see the system control program publication, Data Management Concepts.

An example of the job control statements required to use the VTOC display program to list the contents of the VTOC of a disk storage device follows:

```
// JOB RUNVTOC
// ASSGN SYS004,X'191'          (input)
// ASSGN SYS005,X'00E'          (output)
// EXEC LVTOC
/&
```

To obtain the output on a particular device, it must be assigned to SYS005. The appropriate job control statement are required if labeled magnetic tape or a direct-access device is used. Refer to DOS/VSE DASD Labels for a description of the Format I data set label used in a direct-access storage volume table of contents.

APPENDIX F. REQUIREMENT FOR PROBLEM DETERMINATION AND APAR SUBMISSION

When a member of IBM programming support personnel is called to examine the suspected malfunctioning of an IBM program product, that representative will first determine whether or not the malfunction really is a problem in the program product. If a decision is made that the program product is at fault, a check must then be made to determine whether the fault is a known fault for which an existing fix can be obtained. If the fault is not known, the problem must be referred to the appropriate program maintenance group within IBM for analysis and correction. The process of referring a problem to IBM involves submitting a report known as an APAR (Authorized Program Analysis Report), which must be accompanied by material to enable the program maintenance personnel to analyze the problem.

To enable IBM program maintenance personnel to analyze a problem, it must be possible to reproduce it at the IBM program maintenance center. It will therefore be essential to supply with the APAR the source program to enable the problem to be reproduced and analyzed. Faster resolution of the APAR may be possible if some or all of the material listed in Figure 106 on page 328 is supplied and if the source program is reduced to the smallest, least complex form which still contains the problem.

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 the failure of the PL/I program during execution. Listings derived from separate compilations or executions are of no value and may in fact be misleading to the program support personnel.

Original Source Program

The original PL/I source program must be supplied in a machine-readable form such as a deck of punched cards or a reel of magnetic tape. The copy of the program supplied must be identical to the listing that is also supplied.

Use of the Preprocessor

If the compilation includes preprocessing, the source program submitted should include, either as a card deck or on magnetic tape, the source module obtained by means of the compiler MDECK option.

If the problem is known to have occurred during preprocessing, a listing of the source program being preprocessed must be supplied. If the preprocessing involves the use of the %INCLUDE statement, a copy of the PL/I source statement module(s) included should be supplied in a machine-readable form. If source statement modules are not supplied in the original submission of the APAR, the APAR will not be acted upon until they are supplied.

Job Control Statements

Listings of job control statements used to run the program must be supplied. Where there are a large number of job control statements, supply these also in a machine-readable form such as on punched cards or on magnetic tape. This will assist the program maintenance personnel to reproduce the problem more quickly.

Operating Instructions/Console Log

In the case of an execution-time failure of a program that processes a number of data sets or that operates in a complicated environment, such as a teleprocessing application, it is essential that adequate description of the processing and the environment is given to enable it to be recreated. Although it may be impossible to supply console logs and operating procedures, a complete description of the application, the organization of the data sets, and adequate operating instructions are vital for IBM program support personnel to reproduce the problem.

Listings

A listing of the source program is essential. Other compiler-generated listings, while not essential, may assist in producing a faster resolution of the APAR. If any of the compiler options that must be specified in order to obtain material for submission with an APAR have been deleted at system generation, they can be restored for temporary use by means of the compiler CONTROL option.

There should be no %NOPRINT statements in the listing unless they are pertinent to the problem.

Linkage Editor Map

When a problem occurs at execution time, a linkage editor map is essential. The linkage editor map will be used in the analysis of the storage dump that must also be obtained when the program failed.

Execution-time Dumps

If the problem occurs during execution of the PL/I program, a storage dump must be supplied. A dump can be obtained by using a stand-alone dump program. However, if possible, a formatted PL/I dump produced by the PL/I error-handling facilities should be provided. A PL/I dump is obtained by using the PLIDUMP facility described in Chapter 11.

Compiler Failure under CMS

If the failure occurs while compiling or executing a program or programs under CMS, full details of the Virtual Machine environment must be supplied. This can best be done as follows: immediately prior to invoking the compiler to reproduce the problem, issue the following CMS commands:

```
QUERY SYSNAMES
QUERY SET
QUERY TERMINAL
QUERY VIRTUAL
QUERY SEARCH
QUERY DISK *
QUERY LIBRARY
QUERY DOSLIB
QUERY UPSI
QUERY OPTION
DLBL    (without operands)
LISTIO
```

Invoke the compiler using the DOSPLI command, specifying the option DUMP *PROCESS card that precedes the source program, along with any other options required to produce the relevant output on a line printer.

A listing of the PL/I source program and 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 full details of all input entered and responses received.

Applied PTFs

A list of any program temporary fixes (PTFs) and local fixes applied to either the compiler or its libraries must be supplied.

Submitting the APAR

When submitting material for an APAR to IBM, ensure that any magnetic tapes and decks of punches cards that are supplied containing source programs, job stream data, data sets, or libraries are carefully packed and clearly identified.

Each magnetic tape submitted should have the following information attached and visible:

Material Required	Compiler Option	When Required
Original source program		C, E, M
Job Control Statements		C, E
Operating instructions/ Console log		E
Listings:		
Source listing	SOURCE (S)	C, E, M
Cross-reference listing	XREF (X)	C, E, M
Attribute table	ATTRIBUTES (A)	C, E, M
Aggregate table	AGGREGATE (AG)	C, E, M
Storage table	STORAGE (STG)	C, E, M
Compiler options	OPTIONS (OP)	C, E, M
Object listing	LIST	C, E, M
Compiler termination dump	DUMP (DU)	C, M
Linkage editor map	MAP (linkage editor option)	E
Execution-time dump		E, M
User subroutines		E, M
User data sets or CMS files		E, M
Preprocessor input listing	INSOURCE (IS)	P, M
Preprocessor output	MDECK (MD)	C, E, M
Partition/Region size		C, E, M
Virtual machine configuration		M
List of applied PTFs		C, E, M

Note:
"C" indicates the requirements for a compile-time error;
"E" indicates the requirements for an execution-time error;
"P" indicates the requirements for a processor error;
"M" indicates the requirements for a conversational (CMS) error.

Under CMS, the compiler options must be specified in a *PROCESS card that precedes the source program.

Figure 106. Summary of Requirements for APAR Submission

- The APAR number assigned by IBM
- The contents of the volume (source program job control statements, or data, etc.)
- The recording mode and density
- All relevant information about the labels used for the volume and its data sets
- The record format and blocking sizes used for each data set
- The name of the program that created each data set.

Each card deck submitted must have the following information attached and visible:

- The APAR number assigned by IBM
- The contents of the card deck (source program, job control statements, or data, etc.).

This information will ensure that a magnetic tape or card deck will not be lost if it becomes separated from the rest of the APAR material, and that its contents are readily accessed.

INDEX

Reader's Comment Form

DOS PL/I Optimizing Compiler:
Programmer's Guide
SC33-0008-6

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.*

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.

Note: Staples can cause problems with automated mail sorting equipment.
Please use pressure sensitive or other gummed tape to seal this form.

SC33-0008-6

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

IBM®

DOS PL/I Optimizing Compiler: Programmer's Guide (File No. S360/S370-29) Printed in U.S.A. SC33-0008-6