OS
PL/I Optimizing Compiler:
Programmer's Guide

Program Numbers 5734-PL1
5734-LM4
5734-LM5

(These program products are available as composite package 5734-PL3)
Third Edition (December 1974)

This is a major revision of, and obsoletes, SC33-0006-2. This edition applies to Version 1 Release 2 Modification 2 of the PL/I Optimizing Compiler and to all subsequent releases until otherwise indicated in new editions or Technical Newsletters. Changes will continually be made to the information herein; before using this publication in connection with the operation of IBM systems, consult the latest IBM System/360 and System/370 Bibliography, Order No. 0360-0822-0360, for the editions that are applicable and current.

Changes or additions to the text and figures are indicated by a vertical line to the left of the change. Chapter 4 has been restructured.

Requests for copies of IBM publications should be made to your IBM representative or to the IBM branch office serving your locality.

A form for reader's comments is provided at the back of this publication. If the form has been removed, comments may be addressed to IBM United Kingdom Laboratories Ltd., Programming Publications, Hursley Park, Winchester, Hampshire, England. Comments become the property of IBM.

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

The compiler is designed to operate under Release 20.1 of the IBM Operating System and under all subsequent releases and modifications unless otherwise stated in a revision of the Program Product Specifications. The compiler also operates under Release 1.0 and all subsequent releases of the Conversational Monitor System (CMS) component of the Virtual Machine Facility/370 (VM/370).

An MFT, MVT, VS1, or VS2 version of the Operating System is required. Note that PL/I multitasking facilities can be used only on an MVT or VS2 system.

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

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

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

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

The first three chapters cover basic topics, and are intended primarily for casual (non-specialist) programmers or for newcomers to IBM System/360 or IBM System/370. The reader is assumed to have only an elementary grasp of PL/I and the basic concepts of data processing. These chapters introduce the reader to the operating system, and explain how to run a PL/I program and how to define a 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. Most of this information is equally relevant to the use of the compiler in a batch or TSO environment.

Chapter 4 describes the optimizing compiler, the data sets it requires, its optional facilities, and the listings it produces. Chapter 5 contains similar information for the linkage editor and loader, one of which is needed in addition to the compiler to prepare a PL/I program for execution.

Chapters 6 through 10 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 describes the standard cataloged procedures provided by IBM for the optimizing compiler, and explains how to modify them.

Chapter 12 deals with the facilities available for debugging PL/I programs.

Chapter 13 explains how to link programs written in PL/I with those written in assembler language. (The optimizing compiler implements language designed to facilitate communication between programs written in PL/I and those written in FORTRAN, COBOL, and ASSEMBLER; these facilities are described in the language reference manual listed under "Associated Publications," below.)

Chapters 14 and 15 are concerned with the use of built-in subroutines included in the resident library to provide direct interface between PL/I programs and the operating system sort/merge and checkpoint/restart facilities.

A series of appendixes supply sundry reference information.
Associated Publications

The language implemented by the optimizing compiler is described in the following publication:


For information on how to use the compiler in a TSO environment refer to:

OS Time Sharing Option: PL/I Optimizing Compiler, Order No. SC33-0029

For information on how to use the compiler under the Conversational Monitor System of VM/370, refer to:


The diagnostic messages issued by the compiler and the transient library are listed in the following publication, together with explanations, where necessary, and suggested programmer response:

OS PL/I Optimizing Compiler: Messages, Order No. SC33-0027

Recommended Publications

The following publications are referred to in this programmer's guide. They contain additional details about particular topics discussed in this manual.

OS PL/I Optimizing Compiler: Execution Logic, Order No. SC33-0025

OS Introduction, Order No. GC28-6534

OS Job Control Language Reference, Order No. GC28-6704


OS Linkage Editor and Loader, Order No. GC28-6538

OS System Programmer's Guide, Order No. GC28-6550

OS Utilities, Order No. GC28-6586

OS Sort/Merge, Order No. SC28-6543


OS Supervisor and Data Management Macro Instructions, Order No. GC28-6647

OS Programmer's Guide to Debugging, Order No. GC28-6670

Terminal Commands and Compiler Options: Reference Summary, Order No. SC33-6005

Availability of Publications

The availability of a publication is indicated by its use key, the first letter in the order number. The use keys are

G - General: available to users of IBM systems, products, and services without charge, in quantities to meet their normal requirements; can also be purchased by anyone through IBM branch offices.

S - Sell: can be purchased by anyone through IBM branch offices.
CHAPTER 1: INTRODUCTION .................. 1
The Optimizing Compiler .................. 1
The Operating System .................. 1
Time Sharing Option .................. 1
Jobs and Job Steps .................. 2
Job Control Language .................. 2
Cataloged Procedures .................. 2
Executing a PL/I Program .................. 2

CHAPTER 2: HOW TO RUN A PL/I PROGRAM 5

CHAPTER 3: HOW TO CREATE AND ACCESS 7
A DATA SET .................. 7
Using a Data Set .................. 7
How to Create a Data Set .................. 8
Type of Output Device (UNIT=) .................. 8
Volume Serial Number (VOLUME=SER=) .................. 9
Name of Data Set (DSNAME=) .................. 9
Record Type (DCB=) .................. 9
Auxiliary Storage Required .................. 9
Disposition of Data Set (DISP=) .................. 10
How to Access a Data Set .................. 10
Type of Input Device (UNIT=) .................. 10
Volume Serial Number (VOLUME=SER=) .................. 10
Name of Data Set (DSNAME=) .................. 10
Record Type (DCB=) .................. 10
Auxiliary Storage Required .................. 10
Disposition of Data Set (DISP=) .................. 11
Special-purpose Parameters 11
System Output (SYSOUT=) .................. 11
Data in the Input Stream (* and DATA) .................. 11
Examples .................. 13

CHAPTER 4: THE COMPILER 15
Description of the Compiler 15
Job Control Statements for 17
Compilation 19
EXEC Statement 19
DD Statements for the Standard 19
Data Sets 19
Input (SYSIN, or SYSCIN) 19
Output (SYSLIN, SYSPUNCH) 19
Temporary Workfile (SYSTUT) 20
Listing (SYSPRINT) 20
Source Statement Library (SYSLIB) 20
Example of Compiler JCL 20
Optional Facilities 20
Specifying Compiler Options 21
Specifying Compiler Options in the EXEC Statement 21
Specifying Compiler Options in the PROCESS Statement 22
Compiler Options 22
AGGREGATE Option 22
ATTRIBUTES Option 22
CHARSET Option 22
COMPILE Option 23

Contents
CONTROL Option 24
COUNT Option 24
DECK Option 24
DUMP Option 24
ESD Option 24
FLAG Option 24
FLOW Option 24
GONUMBER Option 25
GOSTMT Option 25
IMPRECISE Option 25
INCLUDE Option 25
INSOURCE Option 25
LINECOUNT Option 25
LIST Option 25
MESSAGE Option 26
MACRO Option 26
MARGIN Option 26
MARGINS Option 26
MDECK Option 26
NAME Option 27
NEST Option 27
NUMBER Option 27
OBJECT Option 27
OFFSET Option 28
OPTIMIZE Option 28
OPTIONS Option 28
SEQUENCE Option 28
SIZE Option 28
SMESSAGE Option 29
SOURCE Option 29
STMT Option 29
STORAGE Option 29
SYNTAX Option 29
TERMINAL Option 30
XREF Option 30
Specifying Execution-Time Options 30
Specifying Execution-Time Options in the FLIXOPT String 30
Specifying Execution-Time Options in the EXEC Statement 31
Execution-Time Options 31
Execution-time Storage 32
Execution-Time COUNT Option 33
Execution-Time FLOW Option 34
Compiler Listing 34
Heading Information 35
Options Used for the Compilation 35
Preprocessor Input 35
Source Program 35
Statement Nesting Level 36
Attribute and Cross-reference Table 36
Attribute Table 36
Cross-reference Table 36
Aggregate Length Table 37
Storage Requirements 37
Statement Offset Table 38
External Symbol Dictionary 38
ESD Entries 39
Other ESD Entries 39

v
Figures

Figure 1-1. A JOB statement .......................... 3
Figure 2-1. How to write a PL/I
program ........................................ 6
Figure 3-1. Information to be
specified when creating a data set ...... 7
Figure 3-2. Creating a CONSECUTIVE
data set: essential parameters of DD
statement ......................................... 8
Figure 3-3. Accessing a CONSECUTIVE
data set: essential parameters of DD
statement ......................................... 11
Figure 3-4. Creating a CONSECUTIVE
data set ........................................... 12
Figure 3-5. Accessing a CONSECUTIVE
data set ........................................... 12
Figure 4-1. Simplified flow diagram
of the compiler ................................. 16
Figure 4-2. Compiler standard data
sets .................................................. 18
Figure 4-3. Record sizes for SYSUT1 20
Figure 4-4. Typical job control
statements for compiling a PL/I
program ............................................. 21
Figure 4-5. Compiler options,
abbreviations, and defaults in batch
mode ............................................... 23
Figure 4-6. Format of the FLAG
option ............................................. 24
Figure 4-7. Compiler listings and
associated options ............................. 35
Figure 4-8. Contents of columns 73
to 80 of source statements .......... 36
Figure 4-9. Standard entries in the
ESD .................................................. 39
Figure 4-10. Selecting the lowest
severity of messages to be printed,
using the PLANG option ................. 40
Figure 4-11. Return codes from
compilation of a PL/I program ........ 41
Figure 4-12. Use of the NAME option
in batched compilation ..................... 42
Figure 4-13. Example of batched
compilation, including execution ...... 42
Figure 4-14. Example of batched
compilation, excluding execution ...... 42
Figure 4-15. Format of the
preprocessor output ......................... 43
Figure 4-16. Using the preprocessor
to create a member of a source
program library ......................... 44
Figure 4-17. Including source
statements from a library ............... 45
Figure 4-18. The sequence of entries
in the ddname list ......................... 46
Figure 5-1. The CSECT IDR
information ..................................... 49
Figure 5-2. Basic linkage editor
processing ..................................... 50
Figure 5-3. Main storage
requirements for linkage editor
IEWLFxxx ..................................... 51
Figure 5-4. Linkage editor standard
data sets ......................................... 52
Figure 5-5. Typical job control
statements for link editing a PL/I
program ......................................... 54
Figure 5-6. Coding the SIZE option ... 55
Figure 5-7. Linkage editor listings
and associated options ................. 55
Figure 5-8. Diagnostic message
severity codes .................................. 56
Figure 5-9. Return codes from the
linkage editor ................................. 57
Figure 5-10. Processing of
additional data sources ................. 59
Figure 5-11. Overlay structure and
its tree ......................................... 60
Figure 5-12. Creating and executing
the overlay structure of Figure 5-11 .... 61
Figure 5-13. Control sections to be
deleted for optimum space-saving .... 63
Figure 5-14. Example of link-editing
a fetchable load module ............... 63
Figure 5-15. Basic loader processing 55
Figure 5-16. Loader processing, link-
pack area and SYSLIB resolution .... 65
Figure 5-17. Main storage
requirements for the loader .......... 66
Figure 5-18. Loader standard data
sets ............................................... 66
Figure 5-19. Job control language
for load-and-go .............................. 68
Figure 5-20. Object and load modules
in load-and-go .............................. 68
Figure 5-21. Contents of SYSLOUT
and SYSPRINT data sets ................. 70
Figure 6-1. A hierarchy of indexes .... 73
Figure 6-2. Fixed-length records .... 75
Figure 6-3. Variable-length records 76
Figure 6-4. The three main types of
data set ......................................... 77
Figure 6-5. The access methods used
by the compiler .............................. 81
Figure 6-6. Access methods for
record-oriented transmission ......... 82
Figure 6-7. How the operating system
completes the DCB ......................... 82
Figure 6-8. Card read punch 2540:
stacker numbers ............................ 83
Figure 6-9. An example of a program
to link edit the DPI ......................... 88
Figure 7-1. Creating a data set:
essential parameters of DD statement 92
Figure 7-2. Creating a data set with
stream-oriented transmission ......... 93
Figure 7-3. Accessing a data set:
essential parameters of DD statement 94
Figure 7-4. Accessing a data set
with stream-oriented transmission .... 94
Figure 7-5. Printer control codes
used by a PRINT file ........................ 95
Figure 7-6. Creating a data set
using a PRINT file ............................ 96
Figure 7-7. Tab control library

The Optimizing Compiler

The PL/I Optimizing Compiler is a processing program that translates PL/I source programs, diagnosing errors as it does so, into IBM System/360 machine instructions. These machine instructions make up an object program. (Later in this chapter there is a description of how an object program is prepared for execution.)

The compiler is designed to produce efficient object programs either with or without optimization. This optimization, which is optional, can be specified by the programmer by means of a compiler option. (See Chapter 4 for details.)

If optimization is specified, the machine instructions generated will be optimized if necessary, to produce a very efficient object program.

If optimization is not specified, compilation time will be reduced.

The optimizing compiler can also be used conversationally. It can be invoked from a remote terminal to compile and execute a PL/I source program, and return the results to the terminal or to a printer.

The optimizing compiler requires a minimum of 50K bytes of main storage when used with MFT and a minimum of 52K when used with MVT. (For minimum storage under OS/VS see Appendix G.) In any case it will work more efficiently with larger amounts of main storage.

The Operating System

The optimizing compiler must be executed through the IBM Operating System. This operating system is used with both System/360 and System/370.

The operating system relieves the programmer of routine and time-consuming tasks by controlling the allocation of storage space and input/output devices. The throughput of the system is increased because the operating system can process a stream of jobs without intervention by the operator.

The operating system comprises a control program and a number of processing programs. The control program supervises the execution of all processing programs, and provides services that are required by the processing programs during their execution. The processing programs include such programs as compilers, the linkage editor, and the loader (described later in this chapter). The operating system is described in the publication OS Introduction.

The optimizing compiler can be used with four operating system control programs:

- MFT (Multiprogramming with a Fixed number of Tasks) permits up to fifteen jobs to be processed concurrently, each job occupying a separate area of main storage termed a partition.
- MVT (Multiprogramming with a Variable number of Tasks) permits up to fifteen jobs to be processed concurrently, each job occupying a separate area of main storage termed a region.
- VS1 and VS2 (Virtual Storage) employ addressable auxiliary storage that appears to the user as main storage. In use VS1 and VS2 are generally similar to MFT and MVT respectively; the differences are explained in Appendix G. Except as explained in the appendix, all information in this manual about MFT applies to VS1, and all information about MVT applies to VS2.

TIME SHARING OPTION

An optional facility of the MVT operating system is the Time Sharing Option (TSO). One or more regions can be allocated to TSO and several users can have concurrent access to the system. Each user enters his jobs from a remote terminal and can receive the results at the terminal. (To contrast it with this "conversational" mode of operation, the more conventional method of submitting jobs through the system operator is called batch operation.)

This programmer's guide forms a complete guide to the use of the optimizing compiler in a batch environment. It also provides essential background and reference information for the TSO user; however, instructions on how to use TSO and how to use the optimizing compiler with TSO are contained in the publications TSO Terminal
JOBS AND JOB STEPS

In a batch environment, the order of processing jobs is determined by a user-defined job class and/or priority. Thus the order in which jobs are processed may differ from the order in which they are entered. Consequently jobs should be independent of each other.

A job comprises one or more job steps, each of which involves the execution of a program. Since job steps are always processed one-by-one in the order in which they appear, they can be interdependent. For example, the output from one job step can be used as the input to a later one, and the processing of a job step can be made dependent on the successful completion of a previous job step.

JOB CONTROL LANGUAGE

Job control language (JCL) is the means by which a programmer defines his jobs and job steps to the operating system; it allows the programmer to describe the work he wants the operating system to do, and to specify the input/output facilities he requires.

Chapter 2, "How to Run a PL/I Program," illustrates the use of JCL statements that are essential for the PL/I programmer. These statements are:

- JOB statement, which identifies the start of a job.
- EXEC statement, which identifies a job step and, in particular, specifies the program to be executed, either directly or by means of a cataloged procedure (described later in this chapter).
- DD (data definition) statement, which defines the input/output facilities required by the program executed in the job step.
- /* (delimiter) statement, which separates data in the input stream from the job control statements that follow this data.

JOB, EXEC, and DD statements have the same format, and Figure 1-1 shows an example of a JOB statement on a punched card. These three statements are identified by the character sequence // in columns 1 and 2. Each statement can contain four fields -- name, operation, operand, and comments -- that are separated by one or more blanks. The name field always starts in column 3.

A full description of job control language is given in the publications OS Job Control Language User's Guide and OS Job Control Language Reference.

Cataloged Procedures

Regularly-used sets of job control statements can be prepared once, given a name, stored in a system library, and the name entered into the catalog for that library. Such a set of statements is termed a cataloged procedure. A cataloged procedure comprises one or more job steps (though it is not a job, because it must not contain a JOB statement). It is included in a job by specifying its name in an EXEC statement instead of the name of a program.

Several IBM-supplied cataloged procedures are available for use with the optimizing compiler. Chapter 11 describes these procedures and how to use them.

EXECUTING A PL/I PROGRAM

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

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

To include the required subroutines from the resident library, the object module must be processed by one of two processing programs, the linkage editor and the loader. Subroutines from the resident library may contain references to other subroutines stored in the OS PL/I Transient Library. The subroutines from the transient library do not become a permanent part of the compiled program; they are loaded into main storage when needed during execution of the PL/I program, and the storage they occupy is released when they are no longer needed.
When using the linkage editor, two further job steps are required after compilation. In the first of these steps, the linkage editor converts the object module into a form suitable for execution, and includes subroutines, referred to by the compiler, from the resident library. The program in this form is called a load module. In the final job step, this load module is loaded into main storage and executed.

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

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

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

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

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

This program uses only punched-card input and printed output. For other forms of input/output refer to Chapter 3. The listing produced includes only the standard default items. Many other items can be included by specifying the appropriate compiler options in the EXEC statement. The compiler listing and all the compiler options are described in Chapter 4. The linkage editor listing and the linkage editor options are described in Chapter 5. Appendix F is a sample PL/I program that includes most of the listing items discussed in these two chapters.

The example in Figure 2-1 uses the cataloged procedure PLIXCLG. Several other cataloged procedures are supplied by IBM for use with the optimizing compiler (for example, for compilation only). The use of these other cataloged procedures is described in Chapter 4.

An alternative method of specifying compiler options is by use of the PROCESS statement, which is described in Chapter 4. An example of a PROCESS statement is:

* PROCESS MACRO,OPT(TIME);
JOB statement

EXAMPLE is the name of the job. You can use any name that does not have more than eight alphameric or national characters; the first character must not be numeric. The job name identifies the job within the operating system; it is essential. The parameters required in the JOB statement depend on the conventions established for your installation.

EXEC statement

PLXCLG is the name of a cataloged procedure supplied by IBM. When the operating system meets this name, it replaces the EXEC statement with a set of JCL statements that have been written previously and cataloged in a system library.

The cataloged procedure contains three procedure steps:

PLI The compiler processes the PL/I program and translates it into a set of machine instructions called an object module.

LKED The linkage editor produces a load module from the object module produced by the compiler.

GO The load module produced by the linkage editor is loaded into main storage and executed.

DD statement

This statement indicates that the statements to be processed in procedure step PLI follow immediately in the card deck.

SYSIN is the name that the compiler uses to refer to the device on which it expects to find this data. (In this case, the device is the card reader, and the data is the PL/I program.)

Delimiter statement

This statement indicates the end of the data (that is, the PL/I program).

DD statement

This statement indicates that the data to be processed by the program (in procedure step GO) follows immediately in the card deck.

Delimiter statement

This statement indicates the end of the data.

Figure 2-1. How to run a PL/I program
### Chapter 3: How to Create and Access a Data Set

<table>
<thead>
<tr>
<th>Information Required</th>
<th>Parameter of DD statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type of output device to which the data set will be transmitted.</td>
<td>UNIT=</td>
</tr>
<tr>
<td>Serial number of the volume (tape reel, disk pack, etc.) that will contain the data set.</td>
<td>VOLUME=SER= (or VOL=SER=)</td>
</tr>
<tr>
<td>Name of the data set.</td>
<td>DSNAME= (or DSN=)</td>
</tr>
<tr>
<td>Type of records in the data set.</td>
<td>DCB= (see appendix A)</td>
</tr>
<tr>
<td>Amount of auxiliary storage required for the data set (direct-access devices only).</td>
<td>SPACE=</td>
</tr>
<tr>
<td>Disposition of the data set on entry to, and at the end of the job step.</td>
<td>DISP=</td>
</tr>
</tbody>
</table>

Figure 3-1. Information to be specified when creating 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 a reel of paper tape; or it can be recorded on magnetic tape or on a direct-access device such as a magnetic disk or drum. A printed listing can also be a data set, but it cannot be read by a program.

Data sets that are created or accessed by PL/I programs must have one of the following types of organization:

- CONSECUTIVE
- INDEXED
- REGIONAL
- Teleprocessing

The items of data in a CONSECUTIVE data set are recorded in the order in which you present them, and can be accessed only in the order in which they were presented or, in the case of magnetic tape, in the reverse order. The items of data in INDEXED and REGIONAL data sets are arranged according to "keys" that you supply when you create the data sets. Teleprocessing data sets are organized as consecutive groups of data items.

This chapter explains how to create and access CONSECUTIVE data sets stored on magnetic tape or on a direct-access device. It is intended to provide an introduction to the subject of data management, and to meet the needs of those programmers who do not require the full input/output facilities of PL/I and the operating system. Chapters 6 through 9 contain a full explanation of the relationship between the data management facilities provided by PL/I and those provided by the operating system, and they explain how to create and access all the types of data sets referred to above.

### Using a Data Set

To create or access a data set, you must not only include the appropriate input and output statements in your PL/I program, but you must also supply certain information to the operating system in a DD statement. A DD statement defines a data set and specifies how it will be handled. The information contained in a DD statement enables the operating system to allocate the necessary auxiliary storage devices, and allows the compiler to use the data management routines of the operating system to transmit data between main storage and auxiliary storage.

The language reference manual for this compiler describes the input and output statements that you will need to use in your PL/I program. Essentially, you must declare a file (explicitly or contextually) and open it (explicitly or implicitly) before you can begin to transmit data. A file is the means provided in PL/I for accessing a data set, and is related to a particular data set only while the file is open; when you close the file, the data set is no longer available to your program. This arrangement allows you to use the same file to access different data sets at different times, and to use different files to access the same data set.
### Parameters of DD Statement

<table>
<thead>
<tr>
<th>Storage Device</th>
<th>Parameters of DD Statement</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct access only</td>
<td>When required: Always</td>
<td></td>
</tr>
<tr>
<td></td>
<td>What you must state:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Output device: UNIT= or SYSOUT=</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Block size: DCB=(BLKSIZE=...)</td>
<td></td>
</tr>
<tr>
<td>Direct access and standard</td>
<td>Data set to be used by another job step but only</td>
<td></td>
</tr>
<tr>
<td>labeled magnetic tape</td>
<td>required by this job</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Disposition: DISP=</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Name of data set: DSN=</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Data set to be kept after end of job</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Volume serial number: VOL=SER=</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Data set to be on particular volume</td>
<td></td>
</tr>
</tbody>
</table>

*Alternatively, you can specify the block size in your PL/I program by using the ENVIRONMENT attribute.*

---

**Figure 3-2. Creating a CONSECUTIVE data set: essential parameters of DD statement**

You must provide a DD statement for each data set that you will use in each job step. If you use the same data set in more than one job step, each job step that refers to this data set must include a DD statement for the data set.

If you are using a cataloged procedure, such as PLIXCLG (described in Chapters 2 and 10), the DD statement for any data set processed by your program must be associated with the appropriate step of the procedure by qualifying the name of the DD statement with the name of the procedure step. For example:

```
//GO.RESULTS DD ...
```

would indicate a DD statement named RESULTS in procedure step GO, as in the example in Figure 3-5. The name of the DD statement is known as its "ddname".

**Type of Output Device (UNIT=)**

You must always indicate the type of output device (for example, magnetic tape or disk drive, card punch, or printer) on which you want to create your data set. Usually the simplest way to do this is to use the UNIT parameter, although for a printer or a card punch it is often more convenient to use one of the special forms of DD statement discussed under "Special-purpose Parameters," later in this chapter.

**How to Create a Data Set**

The information that you should specify when you create a data set is listed in Figure 3-1, which also shows the parameters of the DD statement that you should use.

The following paragraphs discuss the use of these parameters in creating a CONSECUTIVE data set. Figure 3-2 summarizes this discussion, Figure 3-4 is an example of creating this type of data set, and the subparameters of the DCB parameter are described in Appendix A. The job control language reference publication explains how to code a DD statement.

In the UNIT parameter, you can specify either the type number of the unit (for example, 2311 for a disk drive) or the name of a group of devices (for example, SYSDA for any direct-access device). The group names are established for a system during system generation.
A unit of auxiliary storage such as a reel of magnetic tape or a magnetic disk pack is termed a volume; a volume can contain one or more data sets, and a data set can extend to more than one volume. A volume is identified by a serial number that is recorded within it (and usually printed on the label attached to it). Although a deck of cards, a printed listing, and a reel of paper tape can be considered to be volumes, they do not have serial numbers.

Specify a volume serial number only if you want to place the data set in a particular volume. If you omit the VOLUME parameter, the operating system will print in your program listing the serial number of the volume in which it placed the data set.

The VOLUME parameter has several subparameters. To specify a volume serial number, you need only the SER (serial number) subparameter (for example, VOLUME=SER=12345).

NAME OF DATA SET (DSNAME=)

You must name a new data set if you want to keep it for future jobs. If the data set is temporary (required only for the job in which it is created), you can still name it, but you need not; if you omit the DSNAME parameter, the operating system will assume that the data set is temporary, and will give it a temporary name. Alternatively, you can specify your own temporary name by prefixing it with the characters &. For example:

DSNAME=&TEMP

This is especially useful if you want to use the temporary data set in more than one step of your job. The cataloged procedures supplied with the optimizing compiler contain examples of such use.

RECORD TYPE (DCB=)

You can give record-type information either in your PL/I program (in the ENVIRONMENT attribute or LINESIZE option) or in a DD statement. This discussion refers only to the DD statement, and does not apply if you decide to give the information in your program; refer to the language reference manual for this compiler for a description of the ENVIRONMENT attribute and the LINESIZE option.

The type of record in a data set is defined by its format, its physical length (block size), and the length of the subsections (logical record length) which together can be considered to make up a physical record.

The records in a data set must have one of the following formats:

- F fixed length
- V variable length (D- or V-format)
- U undefined length

F-, D-, and V-format records can be blocked (FB, DB, or VB) or unblocked (F, D, or V); V-format records can be spanned. (A spanned record is a record whose length can exceed the size of a block. If this occurs, the record is divided into segments and accommodated in two or more consecutive blocks. D-format indicates that the record is in an ASCII data set. (See the language reference manual for this compiler for details of ASCII data sets.) In most cases, you must specify a block size. If you do not specify a record length, unblocked records of length equal to the block size are assumed. If you are using a PRINT file to produce printed output, you do not need to specify a block size in your DD statement or in your PL/I program; in the absence of other information, the compiler supplies a default line size of 120 characters.

To give record-type information in a DD statement, use the RECFM (record format), BLKSIZE (block size), and LRECL (logical record length) subparameters of the DCB parameter. The DCB parameter passes information to the operating system for inclusion in the data control block, a table maintained by the data management routines of the operating system for each data set in a job step; it contains a description of the data set and how it will be used. If your DCB parameter includes more than one subparameter, you must enclose the list in parentheses. For example:

DCB=(RECFM=FB,BLKSIZE=1000,LRECL=50)

AUXILIARY STORAGE REQUIRED (SPACE=)

When creating a data set on a direct-access device, you must always specify the amount of auxiliary storage that the data set will need. Use the SPACE parameter to specify the number of cylinders, tracks, or blocks.

Chapter 3: How to Create and Access a Data Set
that the data set will need. If you intend to extend the data set in a later job or job step, ensure that your original space allocation is sufficient for future needs; you cannot make a further allocation later. If the SPACE parameter appears in a DD statement for a non-direct-access device, it is ignored.

DISPOSITION OF DATA SET (DISP=)

To keep a data set for use in a later job step or job, you must use the DISP parameter to specify how you want it to be handled. You can pass it to another job step, keep it for use in a later job, or enter its name in the system catalog. If you want to keep the data set, but do not want to include its name in the system catalog, the operating system will request the operator to demount the volume in which it resides and keep it for you. If you omit the DISP parameter, the operating system will assume that the data set is temporary and will delete it at the end of the job step.

The DISP parameter can contain two positional subparameters. The first specifies whether the data set is new or already exists, and the second specifies what is to be done with it at the end of the job step. If you omit the first, you must indicate its absence by a comma. For example:

```
DISP=(,CATLG)
```

specifies that the data set is to be cataloged at the end of the job step. The omission of the first subparameter means that the data set is assumed by default to be new.

How to Access a Data Set

To access (that is, read or update) an existing data set, your DD statement should include information similar to that given when the data set is created. However, for data sets on labeled magnetic tape or on direct-access devices, you can omit several parameters because the information they contain is recorded with the data set by the operating system when the data set is created: Figure 3-3 summarizes the essential information and Figure 3-5 is an example of accessing this type of data set. The subparameters of the DCB parameter are described in Appendix A, and the job control language reference publication explains how to code a DD statement.

Except in the special case of data in the input stream (described under "Special-purpose Parameters," later in this chapter), you must always include the name of the data set (DSNAME) and its disposition (DISP).

TYPE OF INPUT DEVICE (UNIT=)

You can omit the UNIT parameter if the data set is cataloged or if it is created with DISP=(NEW,PASS) in a previous job step of the same job. Otherwise, it must always appear. (PASS specifies that the data set is to be passed for use by a subsequent job step in the same job).

VOLUME SERIAL NUMBER (VOLUME=SER=)

You can omit the VOLUME parameter if the data set is cataloged or if it is created with DISP=(,PASS) in a previous job step of the same job. Otherwise it must always appear.

NAME OF DATA SET (DSNAME=)

The DSNAME parameter can either refer back to the DD statement that defined the data set in a previous job step, or it can give the actual name of the data set. (You would have to use the former method to refer to an unnamed temporary data set.)

RECORD TYPE (DCE=)

You can omit the DCE parameter if the record information is specified in your PL/I program, using the ENVIRONMENT attribute, or if you are accessing a data set on a direct-access device or standard labeled magnetic tape. Otherwise you must specify the DCE parameter for punched cards, paper tape, or unlabeled magnetic tape.

AUXILIARY STORAGE REQUIRED (SPACE=)

You cannot add to, or otherwise modify, the space allocation made for a data set when it is created. Accordingly, the SPACE parameter is never required in a DD statement for an existing data set.
Parameters of DD Statement

<table>
<thead>
<tr>
<th>When required</th>
<th>What you must state</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Always</td>
<td>Name of data set</td>
<td>DSN=</td>
</tr>
<tr>
<td></td>
<td>Disposition of data set</td>
<td>DISP=</td>
</tr>
<tr>
<td>If data set not cataloged</td>
<td>All devices</td>
<td>Input device</td>
</tr>
<tr>
<td></td>
<td>Magnetic tape and volume serial number</td>
<td>VOL=SER=</td>
</tr>
<tr>
<td></td>
<td>Direct access</td>
<td></td>
</tr>
<tr>
<td>For punched cards, paper tape, or unlabeled magnetic tape</td>
<td>Block size</td>
<td>DCB=(BLKSIZE=...)</td>
</tr>
</tbody>
</table>

*Alternatively, you can specify the block size in your PL/I program by using the ENVIRONMENT attribute.

Figure 3-3. Accessing a CONSECUTIVE data set: essential parameters of DD statement

DISPOSITION OF DATA SET (DISP=)

Except for unit record devices (such as card readers), you must always include the DISP parameter to indicate to the operating system that the data set exists. Code DISP=SHR if you want to read the data set, DISP=OLD if you want to read and/or overwrite it, or DISP=MOD if you want to add records to the end of it.

To route your output through a system output device, use the SYSOUT parameter in your DD statement. For example, to punch cards, use the DD statement:

```
//GO.PUNCH DD SYSOUT=B
```

Special-purpose Parameters

Three parameters of the DD statement have special significance because you can use a very simple form of DD statement; they are:

- `SYSOUT=`
- `*`
- `DATA`

`SYSOUT=` is particularly useful for printed or punched-card output, and `*` and `DATA` allow you to include data in the input stream.

Data in the Input Stream (`*` and `DATA`)

A convenient way to introduce data to your program is to include it in the input stream with your job control statements. Data in the input stream must, like job control statements, be in the form of 80-byte records (usually punched cards), and must be immediately preceded by a DD statement with the single parameter `*` in its operand field. For example:

```
//GO.SYSIN DD *
```

To indicate the end of the data, you may optionally include a delimiter job control statement (`/*`). If you omit the `/*` delimiter, the end of the data is determined by the next job control statement (commencing `//` in the first two columns) in the input stream.

If your data includes records that start with `//` in the first two columns use the parameter `DATA`. For example:

Chapter 3: How to Create and Access a Data Set
//OPT3#4 JOB
//STEP1 EXEC PLIXCLG
//PLI.SYSIN DD *
CREATE: PROC OPTIONS(MAIN);

DCL PUNCH FILE STREAM OUTPUT,
  DISK FILE RECORD OUTPUT SEQUENTIAL,
    1 RECORD,
      2(A,B,C,X1,X2) FLOAT DEC(6) COMPLEX;

ON ENDFILE(SYSIN) GO TO FINISH;

NEXT: OPEN FILE(PUNCH), FILE(DISK);
      GET FILE(SYSIN) LIST(A,B,C);
      X1=(-B+SQRT(B**2-4*A*C))/(2*A);
      X2=(-B-SQRT(B**2-4*A*C))/(2*A);
      PUT FILE(PUNCH) EDIT(RECORD) (C(E(16,9)));
      WRITE FILE(DISK) FROM(RECORD);
      GO TO NEXT;
FINISH: CLOSE FILE(PUNCH), FILE(DISK);
END CREATE;

/*
//GO.PUNCH DD SYSPUT=N
//GO.DISK DD DSN=ROOTS,UNIT=2311,VOL=SER=D186,DISP=(NEW,KEEP),
//       SPACE=(TRK,(1,1)),DCB=(RECFM=FB,BLKSIZE=400,LRECL=40)
//GO.SYSIN DD *
5 12 4
4 -10 4
5 16 2
4 -12 10
5 12 9
29 -20 4
*/

Figure 3-4. Creating a CONSECUTIVE data set

//OPT3#5 JOB
//STEP1 EXEC PLIXCLG
//PLI.SYSIN DD *
ACCESS: PROC OPTIONS(MAIN);

DCL RESULTS FILE RECORD INPUT SEQUENTIAL,
  1 RECORD,
    2(A,B,C,X1,X2) FLOAT DEC(6) COMPLEX;

ON ENDFILE(RESULTS) GO TO FINISH;

NEXT: READ FILE(RESULTS) INTO(RECORD);
      PUT FILE(SYSPRINT) SKIP EDIT(RECORD) (C(F(12,2)));
      GO TO NEXT;
FINISH: CLOSE FILE(RESULTS);
END ACCESS;

/*
//GO.RESULTS DD DSN=ROOTS,UNIT=2311,VOL=SER=D186,DISP=(OLD,KEEP)
Figure 3-5. Accessing a CONSECUTIVE data set

12 OS PL/I Optimizing Compiler: Programmer's Guide
In this case, you must always indicate the end of the data by the job control delimiter statement (/*)

Standard Files

PL/I includes two standard files, SYSIN for input and SYSPRINT for output. If your program includes a GET statement without the FILE or STRING option, the compiler uses the file name SYSIN; if it includes a PUT statement without the FILE option, the compiler uses the name SYSPRINT.

If you use one of the IBM-supplied cataloged procedures to execute your program, you will not need to include a DD statement for SYSPRINT; procedure step GO always includes the statement:

//SYSPRINT DD SYSOUT=A

The block size is normally supplied by the compiler; you need not specify it yourself, unless you want blocked output.

If your program uses SYSIN, either explicitly or implicitly, you must always include a corresponding DD statement.

Examples

Two examples of simple applications for CONSECUTIVE data sets are shown in Figures 3-4 and 3-5; both use the cataloged procedure PLIXCLG supplied by IBM.

The first program evaluates the familiar expression for the roots of a quadratic equation and stores the results in a data set on a disk pack and on punched cards. The last but one DD statement (///GO.DISK...) specifies that the newly created data set is to be given the name ROOTS and is to be stored in a volume with serial number D186 on a 2311 Disk Storage Drive. It specifies that fixed-length records, 40 bytes in length, are to be grouped together in blocks, each 400 bytes long. It specifies that the data set is new and that it is to be kept on the volume at the end of the job step; and it specifies that one track of the disk storage drive is to be allocated to the data set with one additional track to be used if more space is required.

The second program accesses the data set on the disk pack created in the first program and prints the results.
This chapter describes the optimizing compiler and the job control statements required to invoke it, and defines the data sets it uses. It describes the compiler options, the listing produced by the compiler, batched compilation, and the preprocessor, all of which are introduced briefly below.

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

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

An object module produced by the compiler is not ready for execution until the appropriate subroutines from the resident library have been included; this is the task of either one of two processing programs, the linkage editor and the loader, described in Chapter 5. An object module that has been processed by the linkage editor is referred to as a load module; an object module that has been processed by the loader is referred to as an executable program.

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

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

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

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

**Description of the Compiler**

The compiler consists of a number of load modules, referred to as phases, each of which can be loaded individually into main storage for execution. A simplified flow diagram is shown in Figure 4-1. The first phase to be loaded is a resident control phase, which remains in main storage throughout compilation. This phase consists of a number of service routines that provide facilities required during execution of the remaining phases. One of these routines communicates with the supervisor program of the operating system for the sequential loading of the remaining phases, which are referred to as processing phases.

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

Each processing phase performs a single function or a set of related functions. Some of these phases must be loaded and executed for every compilation; the requirement for other phases depends on the content of the source program or on the optional facilities selected. Apart from the phases that provide diagnostic information, each phase is executed once only.
Figure 4-1. Simplified flow diagram of the compiler
Input to the compiler is known throughout all stages of the compilation process as text. Initially, this text comprises the PL/I statements of the source program. At the end of compilation, it comprises the machine instructions substituted by the compiler for the source text, together with the inserted references to resident library subroutines for use by the linkage editor or by the loader.

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

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

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

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

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

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

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

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

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

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

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

The external symbol dictionary and the relocation dictionary are described in Chapter 5, which also explains how the linkage editor and the loader use them.

Job Control Statements for Compilation

Although you will probably use cataloged procedures rather than supply all the job control statements required for a job that invokes the compiler, you should be familiar with these statements so that you
### Standard Data Sets

<table>
<thead>
<tr>
<th>Standard dname</th>
<th>Contents of data set</th>
<th>Possible device classes</th>
<th>Record format (RECFM)</th>
<th>Record size (LRECL)</th>
<th>Buffers</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYIN (or SYSCIN)</td>
<td>Input to the compiler</td>
<td>SYSSQ</td>
<td>F,FB,U</td>
<td>&lt;101(100)</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&lt;105(104)</td>
<td></td>
</tr>
<tr>
<td>SYLIN</td>
<td>Object module</td>
<td>SYSSQ</td>
<td>FB</td>
<td>80</td>
<td>2</td>
</tr>
<tr>
<td>SYSPUNCH</td>
<td>Preprocessor output, compiler output</td>
<td>SYSSQ</td>
<td>FB</td>
<td>80</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SYSCP</td>
<td>FB</td>
<td>80</td>
<td>2</td>
</tr>
<tr>
<td>SYSUT1</td>
<td>Temporary workfile</td>
<td>SYSDA</td>
<td>F</td>
<td>1091,1691,3491 or 4051 according to available space</td>
<td>-</td>
</tr>
<tr>
<td>SYSPRINT</td>
<td>Listing, including messages</td>
<td>SYSSQ</td>
<td>VBA</td>
<td>125</td>
<td>2</td>
</tr>
<tr>
<td>SYSLIB</td>
<td>Source statements for preprocessor</td>
<td>SYSDA</td>
<td>F,FB</td>
<td>&lt;101</td>
<td>-</td>
</tr>
</tbody>
</table>

### Notes:

1. The possible device classes are:
   - SYSSQ: Magnetic-tape or direct-access device.
   - SYSDA: Direct-access device.
   - SYSCP: Card-punch device.

2. Any block size can be specified except for SYSLIB and SYSUT1. Block size for SYSLIB depends on the options specified. If the INCLUDE option is specified, the maximum block size is 4260 bytes. If MACRO is specified, for SIZE values below 60K bytes, the maximum is 400 bytes; above 60K bytes, the block size maximum is the value of LRECL for SYSUT1. The block size for SYSUT1 is always provided by the compiler.

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

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

5. The numbers in parentheses in the "Record size" column are the defaults which can be overridden by the user. Where no parentheses are present, the value is fixed and cannot be altered.

---

Figure 4-2. Compiler standard data sets

18 OS PL/I Optimizing Compiler: Programmer's Guide
can make the best use of the compiler, and if necessary, override the statements of the cataloged procedures.

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

- **PLIXC** Compile only.
- **PLIXCL** Compile and link-edit.
- **PLIXCLG** Compile, link-edit, and execute.
- **PLIXCG** Compile, load, and execute.

The following paragraphs describe the essential job control statements for compilation. The IBM-supplied cataloged procedures are described in Chapter 11 and include examples of these statements.

### EXEC STATEMENT

The basic EXEC statement is:

```plaintext
//stepname EXEC PGM=IELOAA
```

The PARM parameter of the EXEC statement can be used to specify one or more of the optional facilities provided by the compiler. These facilities are described under "Optional Facilities," later in this chapter.

### DD STATEMENTS FOR THE STANDARD DATA SETS

The compiler requires several standard data sets, the number depending on the optional facilities specified. You must define these data sets in DD statements with the standard ddnames which are shown, together with other characteristics of the data sets, in Figure 4-2. The DD statements **SYSIN**, **SYSUT1**, and **SYSPRINT** are always required.

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

### Input (SYSIN, or SYSCIN)

Input to the compiler must be a data set defined by a DD statement with the name **SYSIN** or **SYSCIN**; this data set must have CONSECUTIVE organization. The input must be one or more external PL/I procedures; if you want to compile more than one external procedure in a single job or job step, precede each procedure, except possibly the first, with a PROCESS statement (described under "Batched Compilation," later in this chapter).

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

### Output (SYSLIN, SYSPUNCH)

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

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

Chapter 4: The Compiler 19
Temporary Workfile (SYSUT1)

The compiler requires a data set for use as a temporary workfile. It is defined by a DD statement with the name SYSUT1, and is known as the spill file. It must be on a direct-access device. The spill file is used as a logical extension to main storage and is used by the compiler and by the preprocessor to contain text and dictionary information.

Four record sizes are given in Figure 4-2 for SYSUT1. For storage devices other than the 3330, the first three sizes correspond to the amount of storage available to the compiler, as shown in Figure 4-3.

<table>
<thead>
<tr>
<th>Storage</th>
<th>Record Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>50-55K</td>
<td>1091</td>
</tr>
<tr>
<td>56-69K</td>
<td>1691</td>
</tr>
<tr>
<td>over 69K</td>
<td>3491</td>
</tr>
</tbody>
</table>

Figure 4-3. Record sizes for SYSUT1

A record size of 4051 is used on the 3330.

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

Dedicated Data Sets: If a job being run under MVT has several job steps, and each job step requires a data set for use as a temporary workfile, the result is a considerable overhead in time and space. To reduce this as far as possible, you can use dedicated data sets. These are data sets that are created by the operating system when the job is selected for processing. They can be used by each job step that requires a temporary workfile. Dedicated data sets are normally allocated by the initiator and deleted when it terminates. More information on using dedicated data sets is given in Chapter 11.

Listing (SYSPRINT)

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

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

//SYSPRINT DD SYSUT1=A

The compiler always reserves 258 bytes of main storage (129 bytes each) for two buffers for this data set; however, you may specify a block size of more than 129 bytes, provided that sufficient main storage is available to the compiler. (See the discussion of the SIZE option under "Optional Facilities," later in this chapter.)

Source Statement Library (SYSLIE)

If you use the preprocessor %INCLUDE statement to introduce source statements into the PL/I program from a library, you can either define the library in a DD statement with the name SYSLIE, or you can choose your own ddname (or ddnames) and specify a ddname in each %INCLUDE statement. (See "Compile-time Processing," later in this chapter.)

EXAMPLE OF COMPILER JCL

A typical sequence of jcl compiler statements for compiling a PL/I program is shown in Figure 4-4. The DECK and NCOBJECT compiler options, described below, have been specified to obtain an object module as a card deck only. Jcb control statements for link editing an object module in the form of a card deck are shown in Chapter 5.

Optional Facilities

The compiler provides a number of optional facilities, both at compile time and at
Figure 4-4. Typical job control statements for compiling a PL/I program

execution time. Options that can be specified at compile time are known as compiler options. Options that can be specified at execution time are known as execution-time options.

Execution-time and compiler options, their abbreviated forms, and their defaults (as supplied by IBM) are shown in Figures 4-5 and 4-7. An installation can modify or delete defaults according to local requirements; check for any modified defaults at your installation. Deleted compiler options can be reinstated for a compilation by means of the CONTRCL compiler option.

Also provided is the ability to pass an argument to the PL/I main procedure. This facility is described in the section "Specifying Execution-Time Options in the EXEC Statement," later in this chapter.

SPECIFYING COMPILER OPTIONS

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

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

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

To specify options in the EXEC statement, code PARM= followed by the list of options, in any order (except that CONTROL, if used, must be first) separating the options with commas and enclosing the list within single quotation marks, for example:

```
//STEP1 EXEC PGM=IELOAA,FARM='OBJECT,LIST'
```

Any option that has quotation marks, for example PARMIN('c'), must have the quotation marks duplicated. The length of the option list must not exceed 256 characters, including the separating commas (note that only the first 100 characters are printed out on the listing). However, many of the options have an abbreviated form that you can use to save space. If you need to continue the statement onto another line, you must enclose the list of options in parentheses (instead of in quotation marks) encode the options list on each line in quotation marks, and ensure that the last comma on each line except the last is outside of the quotation marks. An example covering all the above points is as follows:

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

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

```
//STEP1 EXEC FLIXCLG,PARM.FLI='A,LIST,ESD'
```
Specifying Compiler Options in the PROCESS Statement

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

```
* PROCESS options;
```

where "options" is a list of compiler options. The list of options must be terminated with a semicolon and should not extend beyond the default right-hand source margin. The asterisk must appear in the first byte of the record (card column 1), and the keyword PROCESS may follow in the next byte (column) or after any number of blanks. Option keywords must be separated by a comma and/or at least one blank.

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

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

```
* PROCESS;
```

Should it be necessary to continue the PROCESS statement onto the next card or record, terminate the first part of the list after any delimiter, up to the default right-hand margin, and continue on the next card or record. Option keywords or keyword arguments may be split, if required, when continuing onto the next record, provided that the keyword or argument string terminates in the right-hand source margin, and the remainder of the string starts in column 1 of the next record. A PROCESS statement may be continued in several statements, or a new PROCESS statement started.

COMPILER OPTIONS

The compiler options are of the following types:

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

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

3. A combination of 1 and 2 above.

The following paragraphs describe the options in alphabetic order. For those options that specify that the compiler is to list information, only a brief description is included; the generated listing is described under "Compiler Listing," later in this chapter.

Figure 4-5 lists all the compiler options with their abbreviated forms and their standard default values for batch mode. Defaults under TSO are given in the TSO User's Guide for this compiler.

AGGREGATE Option

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

ATTRIBUTES Option

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

CHARSET Option

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

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

BCD or EBCDIC: If the source program is written in BCD, specify CHARSET(EBCD); if it is written in EBCDIC, specify CHARSET(EBCDIC). The language reference manual for this compiler lists the EBCDIC
<table>
<thead>
<tr>
<th>Compiler Option</th>
<th>Abbreviated Name</th>
<th>IBM Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>AGGREGATE</td>
<td>NOAGGREGATE</td>
<td>A</td>
</tr>
<tr>
<td>ATTRIBUTES</td>
<td>NOATTRIBUTES</td>
<td>A</td>
</tr>
<tr>
<td>CHARSET(48</td>
<td>60</td>
<td>EBCDIC</td>
</tr>
<tr>
<td>COMPIL</td>
<td>NOCOMPIL</td>
<td>W</td>
</tr>
<tr>
<td>CONTROL('password')</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COUNT</td>
<td>NOCOUNT</td>
<td>C</td>
</tr>
<tr>
<td>DECK</td>
<td>NODECK</td>
<td>D</td>
</tr>
<tr>
<td>DUMP</td>
<td>NODUMP</td>
<td>D</td>
</tr>
<tr>
<td>ESD</td>
<td>NOESD</td>
<td>E</td>
</tr>
<tr>
<td>FLOW(n,m)</td>
<td>NOFLOW</td>
<td>F(1W</td>
</tr>
<tr>
<td>GONUMBER</td>
<td>NOGONUMBER</td>
<td>G</td>
</tr>
<tr>
<td>GOSTMT</td>
<td>NOGOSTMT</td>
<td>G</td>
</tr>
<tr>
<td>IMPRECISE</td>
<td>NOIMPRECISE</td>
<td>I</td>
</tr>
<tr>
<td>INCLUDE</td>
<td>NOINCLUDE</td>
<td>I</td>
</tr>
<tr>
<td>INSOURCE</td>
<td>NOINSOURCE</td>
<td>I</td>
</tr>
<tr>
<td>LINECOUNT(n)</td>
<td></td>
<td>LINECOUNT(n)</td>
</tr>
<tr>
<td>LIST1(n,m)</td>
<td>NOLIST</td>
<td>LMSG</td>
</tr>
<tr>
<td>MACRO</td>
<td>NOMACRO</td>
<td>M</td>
</tr>
<tr>
<td>MAP</td>
<td>NOMAP</td>
<td>M</td>
</tr>
<tr>
<td>MARGINI('c')</td>
<td>NOMARGINI</td>
<td>MI('c')</td>
</tr>
<tr>
<td>MARGINS(m,n,</td>
<td>c)</td>
<td>MAR(m,n,</td>
</tr>
<tr>
<td>MDECK</td>
<td>NOMDECK</td>
<td>M</td>
</tr>
<tr>
<td>NAME('name')</td>
<td>N('name')</td>
<td></td>
</tr>
<tr>
<td>NEST</td>
<td>NONEST</td>
<td>N</td>
</tr>
<tr>
<td>NUMBER</td>
<td>NONUMBER</td>
<td>N</td>
</tr>
<tr>
<td>OBJECT</td>
<td>NOOBJECT</td>
<td>O</td>
</tr>
<tr>
<td>OFFSET</td>
<td>NOOFFSET</td>
<td>O</td>
</tr>
<tr>
<td>OPTIMIZE(TIME</td>
<td>0</td>
<td>2)</td>
</tr>
<tr>
<td>OPTIONS</td>
<td>NOOPTIONS</td>
<td>O</td>
</tr>
<tr>
<td>SEQUENCE(m,n)</td>
<td>NOSEQUENCE</td>
<td>SEQ(m,n)</td>
</tr>
<tr>
<td>SIZE([-]</td>
<td>yyyyyy</td>
<td>[-]</td>
</tr>
<tr>
<td>SOURCE</td>
<td>NOSOURCE</td>
<td>S</td>
</tr>
<tr>
<td>STM</td>
<td>NOSTMT</td>
<td>S</td>
</tr>
<tr>
<td>STORAGE</td>
<td>NOSTORAGE</td>
<td>S</td>
</tr>
<tr>
<td>SYNTAX</td>
<td>NOSYNTAX</td>
<td>W</td>
</tr>
<tr>
<td>TERMINAL([opt-list])</td>
<td>NOTERMINAL</td>
<td>TERM([opt-list])</td>
</tr>
<tr>
<td>XREF</td>
<td>NOXREF</td>
<td>X</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>------------------</td>
<td>---------------------------</td>
</tr>
</tbody>
</table>

Figure 4-5. Compiler options, abbreviations, and defaults in batch mode

representation of both the 48-character set and the 60-character set.

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

**COMPIL**e Option

The **COMPIL**e option specifies that the compiler is to compile the source program unless an unrecoverable error was detected during preprocessing or syntax checking. The **NOCOMPIL**E 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:

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

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

Chapter 4: The Compiler 23
NOCOMPILE(S) No compilation if a severe error or unrecoverable error is detected.

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

DUMP Option

The DUMP option specifies that the compiler is to produce a formatted dump of main storage if the compilation terminates abnormally (usually due to an I/O error or compiler error). This dump is written on the data set associated with SYSPRINT.

CONTROL Option

The CONTROL option specifies that any compiler options deleted for your installation are to be available for this compilation. You must still specify the appropriate keywords to use the options. The CONTROL option must be specified with a password that is established for each installation; use of an incorrect password will cause processing to be terminated. The CONTROL option, if used, must be specified first in the list of options. It has the format:

```
CONTROL('password')
```

where "password" is a character string, not exceeding eight characters.

ESD Option

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

FLAG Option

The FLAG option specifies the minimum severity of error that requires a message to be listed in the compiler listing. The format of the FLAG option is given in Figure 4-6.

```
FLAG(I) List all messages.
FLAG(W) List all except informative messages. If you specify FLAG, FLAG(W) is assured.
FLAG(E) List all except warning and informative messages.
FLAG(S) List only severe error and unrecoverable error messages.
```

Figure 4-6. Format of the FLAG option

FLOW Option

The FLOW option specifies that the compiler is to produce code to enable the flow of control to be listed when the program is executed. The format of the FLOW option is:

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

where "n" is the maximum number of entries to be included in the lists. It should not exceed 32768.

"m" is the maximum number of procedures for which the lists.
are to be generated. It should not exceed 32768.

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

The output produced by the FLOW option is described under "Execution-Time FLOW Option" later in this chapter.

GONUMBER Option

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

GOSTMT Option

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

IMPRECISE Option

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

INCLUDE Option

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

INSOURCE Option

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

LINECOUNT Option

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

\[
\text{LINECOUNT}(n)
\]

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

LIST Option

The LIST option specifies that the compiler is to include a listing of the object module (in a form similar to IBM System/360 assembler language instructions) in the compiler listing. The format of the list option is:

\[
\text{LIST}([m,n])
\]

where \(m\) is the number of the first source statement for which an object listing is required and \(n\) is the number of the last source statement for which an object listing is required. If \(m\) is omitted,
only statement "m" is listed. If the option NUMBER applies, "m" and "n" must be specified as line numbers.

LMESSAGE Option

The LMESSAGE and SMESSAGE options specify that the compiler is to produce messages in a long form (specify LMESSAGE) or in a short form (specify SMESSAGE). Short messages can have advantages in a TSO environment due to the comparatively slow printing speed of a terminal.

MACRO Option

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

MAP Option

The MAP option specifies that the compiler is to produce tables showing the organization of the static storage for the object module. These tables consist of a static internal storage map and the static external control sections. The MAP option is normally used with the LIST option.

MARGINI Option

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

The MARGINI option has the format:

MARGINI('c')

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

MARGINS Option

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

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

MARGINS(m,n,[c])

where "m" is the column number of the left-hand margin. It should not exceed 100.
"n" is the column number of the right-hand margin. It should be greater than m, but not greater than 100.
"c" is the column number of the ANS printer control character. It should not exceed 100 and should be outside the values specified for m and n. Only the following control characters can be used:

(blank) Skip one line before printing.
0   Skip two lines before printing.
-   Skip three lines before printing.
+   No skip before printing.
1   Start new page.

The standard IBM-supplied default for fixed-length records is MARGINS(2,72,0); that for variable-length and undefined-length records is MARGINS(10,100,0). A zero value for "c" specifies that there is no printer control character.

MDECK Option

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

NAME Option

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

It is required if you want the linkage editor to create more than one load module from the object modules produced by batched compilation (see later in this chapter).

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

```
NAME('name')
```

where "name" has from one through eight characters, and begins with an alphabetic character. The linkage editor NAME statement is described in Chapter 5.

NEST Option

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

NUMBER Option

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

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

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

- First 8 columns for undefined-length or variable-length source input records. In this case, 8 is added to the values used in the MARGINS option.
- Last 8 columns for fixed-length source input records.

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

The line number is calculated from the five right-hand characters of the sequence number (or the number specified, if less than five). These characters are converted to decimal digits if necessary. Each time a sequence number is found that is not greater than the preceding line number, a new line number is formed by adding the minimum integral multiple of 100,000 necessary to produce a line number that is greater than the preceding one. The maximum line number permitted by the compiler is 134,000,000; numbers that would normally exceed this are set to this maximum value. Only eight digits are printed in the source listing; line numbers of 100,000,000 or over will be printed without the leading "1" digit.

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

OBJECT Option

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

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

A method of determining statement or line numbers from the offsets given in error messages is given under the heading "Statement Offset Addresses" later in this chapter.

OPTIMIZE Option

The OPTIMIZE option specifies the type of optimization required:

- **NOOPTIMIZE** specifies fast compilation speed, but inhibits optimization for faster execution and reduced main storage requirements.
- **OPTIMIZE(TIME)** specifies that the compiler is to optimize the machine instructions generated to produce a very efficient object program. A secondary effect of this type of optimization can be a reduction in the amount of main storage required for the object module. The use of OPTIMIZE(TIME) could result in a substantial increase in compile time over NOOPTIMIZE.
- **OPTIMIZE(0)** is the equivalent of NOOPTIMIZE.
- **OPTIMIZE(2)** is the equivalent of OPTIMIZE(TIME).

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

OPTIONS Option

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

SEQUENCE Option

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

```
SEQUENCE(m,n)
```

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

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

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

SIZE Option

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

- **SIZE(yyyyyyyy)** specifies that yyyyyyyyy bytes of main storage are to be requested. Leading zeros are not required.
- **SIZE(yyyyyK)** specifies that yyyyyK bytes of main storage are to be requested (1K=1024). Leading zeros are not required.
- **SIZE(-yyyyyy)** specifies that the compiler is to obtain as much main storage as it can, and then
release yyyy bytes to the operating system. Leading zeros are not required.

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

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

The IBM default, and the most usual value to be used, is **SIZE(MAX)**, which permits the compiler to use as much main storage in the partition or region as it can.

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

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

Under MFT, the compiler will operate in a partition of 50K bytes or more of main storage, using its default values for file specifications. Under MVT, a region of 52K bytes or more is required.

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

If **SIZE(MAX)** is specified, it obtains all space remaining in the region or partition (after allowance for subsequent data management storage areas). If a limit is specified then this amount is requested. If the amount available is less than specified, but is more than the minimum workspace required, compilation proceeds. If insufficient storage is available, compilation is terminated. This latter situation should arise only if the region or partition is too small, that is, less than 50K, or if too much space for buffers has been requested. The value cannot exceed the main storage available for the job step and cannot be changed after processing has begun.

This means, that in a batched compilation, the value established when the compiler is invoked cannot be changed for later programs in the batch. Thus it is ignored if specified in a PROCESS statement.

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

**SMESSAGE Option**

See LMESSAGE option.

**SOURCE Option**

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

**STMT Option**

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

**STORAGE Option**

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

**SYNTAX Option**

The **SYNTAX** option specifies that the compiler is to continue into syntax checking after initialization (or after preprocessing if the MACRO option applies) unless an unrecoverable error is detected.

Chapter 4: The Compiler 29
The NOSYNTAX option without an argument causes processing to stop unconditionally after initialization (or preprocessing). With an argument, continuation depends on the severity of errors detected so far, as follows:

- **NOSYNTAX(W)**: No syntax checking if a warning, error, severe error, or unrecoverable error is detected.
- **NOSYNTAX(E)**: No syntax checking if an error, severe error, or unrecoverable error is detected.
- **NOSYNTAX(S)**: No syntax checking if a severe error or unrecoverable error is detected.

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

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

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

### TERMINAL Option

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

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

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

### SPECIFYING EXECUTION-TIME OPTIONS

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

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

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

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

The PLIXCPT string is ignored in a Checkout Compiler/Optimizing Compiler mixture environment.
Specifying Execution-Time Options in the EXEC Statement

The method of coding the PARM parameter in an EXEC statement is given under the heading "Specifying Compiler Options in the EXEC Statement" earlier in this chapter.

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

```
//STEP EXEC PLIXCIG,PARM.GO=('ISA(10K)', REPORT)
```

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

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

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

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

EXECUTION-TIME OPTIONS

The following paragraphs describe the execution-time options, which can be specified in the EXEC statement or in the PLIXOPT string.

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

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

**FLOW**(n,m) specifies that a list of the most recent transfers of control in the execution of the program is to be generated. This option is discussed in greater detail under the heading "Execution-Time FLOW Option" later in this chapter.

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

**ISASIZE** specifies the amount of main storage initially acquired by the PL/I program at execution time.

This storage is known as the initial storage area (ISA). The option has the format:

```
ISASIZE([x],[y],[z])
```

where "x" is the initial storage allocation for the major task, and where "y" is the initial storage allocation for each subtask within the total storage available to the compiler. This value can be used in a multitasking program to prevent a new storage request (with its accompanying time overhead) each time a block is entered during the execution of the subtasks. If you specify enough storage for a whole subtask, these additional requests are not made, and where "z" is the maximum number of subtasks that will be active at any one time.

All storage values must be in bytes or K bytes. If "x" is omitted and "y" or "z" is specified, or if "y" is omitted and "z" is specified, then the separating commas must be used to indicate that a value is missing.

If the multitasking arguments ("y" and "z") are specified for a program that was linked without the multitasking library, they will be ignored, and a diagnostic message will be issued.

The ISA is used for the dynamic allocation of the main storage required by PL/I blocks as they are entered, and by controlled and based variables as they are allocated. If the ISA is large enough to contain these blocks PL/I storage handling will not acquire any more storage from the system.

If ISASIZE is not specified, then in a non-multitasking environment the IBM default value is calculated as follows:

```
(m - n)/2
```

where "m" is the region size for the GC step, and "n" is the load module length. This value is rounded up to a 2K boundary. In a multitasking environment, the default is 8K bytes for the major task and 8K bytes for each subtask. The default value for

Chapter 4: The Compiler 31
the maximum number of active subtasks is 20.

Note that if an initial storage allocation is too large, that is, most variables in STATIC and few controlled and based allocations, there will be a considerable amount of wasted main storage in the ISA. In some cases this may cause the program to terminate abnormally, because there is insufficient storage available for dynamically-loaded modules and for data areas required by the operating system.

If the initial storage allocation is too small, then dynamic main storage requirements will be less efficiently met by individual requests to the system. Furthermore, the defaults may not appear in the storage report given by the REPORT option. If a default ISASIZE is used initially and proves to be too small, then the ISASIZE finally used, which appears in the report, may be less than the default. For instance in a multitasking environment the major task's ISASIZE will default to 8K bytes. If this task has an AUTOMATIC character string variable of say 20K bytes the ISA will be too small. Certain control blocks are placed first in the ISA and occupy about 1K bytes of it. The DSA, which will be greater than 20K because it includes storage for automatic variables, is then allocated. Since there is no room for it in the ISA, more storage is made available from the remainder of the region. However to avoid wastage of storage the unused part of the ISA is freed beforehand. Thus the length of the initial storage allocation will be approximately 1K bytes and this length will be printed in the report.

The execution-time option REPORT is available to enable the programmer to determine exactly what his storage requirements are, apart from I/O requirements.

REPORT specifies that a report of certain program management activity is to be printed. The report will be automatically output to a data set with the dname PLIDUMP or PLIDUMP on program termination. This includes, for example, the amount of storage that was specified in the ISASIZE option, the length of the initial storage area, and the amount of PL/I storage required. This option may be abbreviated to R. The use of the report is described in "Execution-time Storage Requirements", below.

NOREPORT specifies that a report is not required. This option may be abbreviated to NR.

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

NOSTAE specifies that on program initialization, a STAE macro instruction is not to be issued.

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

NOSPIE specifies that on program initialization, a SPIE macro instruction is not to be issued. This option must not be used if extended precision variables are used in the PL/I source program.

The execution-time options are discussed in greater detail in the publication OS PL/I Optimizing Compiler: Execution Logic.

Execution-time Storage Requirements

At execution time there are three separate areas of main storage.

The first area is the load module. Its length can be obtained from the linkage editor output listing.

The second area is the initial storage area (ISA). Its length can be specified by the ISASIZE execution-time option or supplied by default. If supplied by default in a non-multitasking environment, it will be approximately half of the main storage available after the load module has been loaded. In a multitasking environment, it will be 8K bytes for the major task and 8K bytes for each subtask. The ISA will include:

- Dynamic block requirements. These lengths can be obtained from the table produced by the STORAGE compiler option.
• Variable data areas, that is, varying-length strings and arrays, whose bounds or dimensions are not known at compile time. The programmer must calculate these lengths himself.

• Controlled and based variables. These lengths should be known to the programmer.

The third area consists of the remainder of main storage. It is retained by the system and is made available on specific main storage requests for overflow from the ISA and for I/O requirements, that is, file control blocks, buffers, system I/O modules and also for PL/I transient library modules (that is, storage overflow, program initialization, and I/O transmission modules).

The storage requirements in this third area can be calculated with difficulty. The simplest way is to use the Storage Management Facilities (SMF) as described in the publication OS Introduction to determine the total main storage requirements for the job. This figure is only meaningful if an accurate figure for the ISA has been supplied.

The length of the ISA can greatly affect the performance of the program. If it is too large there will be wasted storage in the ISA which might result in insufficient main storage being available for I/O requirements and transient library modules requirements.

If it is too small then dynamic main storage requirements will be met by specific requests to the system (that is, from the third area of main storage) resulting in slow execution. The programmer's total ISA requirements can be determined either by calculation or by using the REPORT execution-time option.

This can most easily be done in one of two ways:

1. If sufficient main storage is available, specify an ISASIZE larger than will be required. The report will then give the amount of this ISA used and this figure will be the optimum ISASIZE.

2. If there is a shortage of main storage specify an ISASIZE of 1, which will ensure that the program will run if at all possible and the report will still give the amount of main storage which should be allocated to the ISA.

Execution-Time COUNT Option

Statement count information can be obtained at execution time only if one of the compiler options COUNT or FLOW was specified at compile time (see "Compiler Options" earlier in this chapter.) If FLOW but not COUNT was specified at compile time, COUNT must be specified at execution to obtain count information. If COUNT was specified at compile time, count information will be produced unless NOCOUNT is specified at execution time.

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

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

```
PROCEDURE name1
  FROM TO COUNT
    1  20  1
    21 30  10
      .  .  
    200 210 1

PROCEDURE name2
  FROM TO COUNT
    1  10  5
      .  .  
      .  .  
```

Three such columns are printed per page.

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

The count tables are printed when the program terminates. If a procedure is invoked with one of the multitasking options, the count table for the invocation is printed when the task terminates.
If an invocation is terminated as a result of the termination of another task, its count table cannot be printed, because it is impossible to determine the point at which it terminated. In these circumstances, only the count table for the first task to terminate can be printed. For example, although a STOP statement will cause all tasks to be terminated, only the count table for the task containing the statement will be printed.

**Execution-Time FLOW Option**

Flow information can be obtained at execution time only if one of the compiler options COUNT or FLOW was specified at compile time (see "Compiler Options" earlier in this chapter.) If FLOW was not specified at compile time, it must be specified at execution time to obtain flow information. If FLOW was specified at compile time, flow information will be produced unless NOFLOW is specified at execution time.

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

```
FLOW(n,m)
```

If "n" and "m" are not specified at execution time, the values taken are as follows:
- If FLOW was specified or defaulted at compile time, the values of "n" and "m" specified or defaulted at compile time are taken.
- If NOFLOW was specified or defaulted at compile time, the IBM default values, (25,10), are taken.

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

The format of each line of flow output is:

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

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

```
sn2
```

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

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

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

**Compiler Listing**

During compilation, the compiler generates a listing, most of which is optional, that contains information about the source program, the compilation, and the object module. It places this listing in the data set defined by the DD statement with the name SYSPRINT (usually output to a printer). In a TSO environment, you can also request a listing at your terminal (using the TERMINAL option). The following description of the listing refers to its appearance on a printed page.

An example of the listing produced for a typical PL/I program is given in appendix F.

Figure 4-7 specifies the components that can be included in the compiler listing, and the order in which they appear. The rest of this section then describes these in detail.

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

System information will appear before and after the listings for each job step if these items use the same output class as the processing programs. The output class for system information is specified in the MSGCLASS parameter of the JOB statement. The level of information produced is specified in the MSGLEVEL parameter.

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

<table>
<thead>
<tr>
<th>Options used for the compilation</th>
<th>Options required</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preprocessor input</td>
<td>OPTIONS</td>
</tr>
<tr>
<td>Source program</td>
<td>MACRO and INSOURCE</td>
</tr>
<tr>
<td>Statement nesting level</td>
<td>SOURCE</td>
</tr>
<tr>
<td>Attribute table</td>
<td>NEST</td>
</tr>
<tr>
<td>Cross-reference table</td>
<td>ATTRIBUTES</td>
</tr>
<tr>
<td>Aggregate length table</td>
<td>XREF</td>
</tr>
<tr>
<td>Storage requirements</td>
<td>AGGREGATE</td>
</tr>
<tr>
<td>Statement offset addresses</td>
<td>STORAGE</td>
</tr>
<tr>
<td>External symbol dictionary</td>
<td>SOURCE, OFFSET, NOSTMT</td>
</tr>
<tr>
<td>Static internal storage map</td>
<td>ESD</td>
</tr>
<tr>
<td>Object listing</td>
<td>MAP</td>
</tr>
<tr>
<td>Messages</td>
<td>LIST</td>
</tr>
<tr>
<td></td>
<td>FLAG</td>
</tr>
</tbody>
</table>

**Figure 4-7. Compiler listings and associated options**

**HEADING INFORMATION**

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

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

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

**OPTIONS USED FOR THE COMPILATION**

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

**PREPROCESSOR INPUT**

If both the options MACRO and INSOURCE apply, the input to the preprocessor is listed, one record per line, each line numbered sequentially at the left.

If the preprocessor detects an error, or the possibility of an error, it prints a message on the page or pages following the input listing. The format of these messages is exactly as described for the compiler messages described under "Messages," later in this chapter.

**SOURCE PROGRAM**

If the option SOURCE applies, the input to the compiler is listed, one record per line; if the input records contain printer control characters or $S$KIP or $S$PAGE statements, the lines will be spaced accordingly.

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

If the option STMT applies, the statements in the source program are numbered sequentially by the compiler, and the number of the first statement in the line appears to the left of each line in which a statement begins. When an END statement closes more than one group or block, all the implied END statements are included in the count. For example:
PROCEDURE;
BEGIN;
IF A=B
THEN A=1;
ELSE D=E;
END;

If the source statements are generated by
the preprocessor, columns 73-80 contain
diagnostic information, as shown in Figure
4-8.

<table>
<thead>
<tr>
<th>Column</th>
<th>Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>73-77</td>
<td>Input line number from which the source statement is generated. This number corresponds to the line number in the preprocessor input listing.</td>
</tr>
<tr>
<td>78,79</td>
<td>Two-digit number giving the maximum depth of replacement by the preprocessor for this line. If no replacement occurs, the columns are blank.</td>
</tr>
<tr>
<td>80</td>
<td>&quot;E&quot; signifying that an error has occurred while replacement is being attempted. If no error has occurred, the column is blank.</td>
</tr>
</tbody>
</table>

Figure 4-8. Contents of columns 73 to 80 of source statements

STATEMENT NESTING LEVEL

If the option NEST applies, the block level and the do-level are printed to the right of the statement or line number under the headings LEV and NT respectively, for example:

<table>
<thead>
<tr>
<th>STMT</th>
<th>LEV</th>
<th>NT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>A:PROC OPTIONS(MAIN);</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>B:PROC(L);</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>DO I=1 to 10;</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>DO J=1 TO 10;</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>X(I,J)=N;</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>END;</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>BEGIN;</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>X=Y;</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>END;</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>END B;</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>END A;</td>
</tr>
</tbody>
</table>

ATTRIBUTE AND CROSS-REFERENCE TABLE

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

/*STRUCTURE*/

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

Attribute Table

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

The attributes INTERNAL and REAL are never included; they can be assumed unless the respective conflicting attributes, EXTERNAL and COMPLEX, appear.

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

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

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

Cross-reference Table

If the cross-reference table is combined with the attribute table, the numbers of the statements or lines in which an identifier appears follow the list of attributes for the identifier. The number
of a statement in which an implicitly-pointer qualified based variable appears will be included not only in the list of statement numbers for that variable, but also in the list of statement numbers for the pointer associated with it implicitly.

If a based variable is referenced without explicit pointer qualification, a reference to the implicit pointer used will be included in the cross-reference listing.

Note that an END statement that refers to a label does not have its statement number listed in the entry for the label.

Identifiers that are initialized during execution of prologue code on entry to a block will have the PROCEDURE or BEGIN statement number included in the list of statement numbers. For example, automatic variables with the INITIAL attribute in a single-block program will have a reference to statement number 1 in the cross-reference table.

The order in which the statement numbers appear for a particular identifier is subject to any reordering of blocks that has occurred during compilation. In general, the statement numbers for the outermost block are given first, followed on the next line by the statement numbers for the inner blocks.

The PL/I text is expanded to a certain extent before the cross-reference list is produced. Consequently, an identifier within a statement may acquire multiple references to the same statement number. Common examples are the use of do-groups and statements involving aggregates.

AGGREGATE LENGTH TABLE

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

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

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

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

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

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

The length of an aggregate may not be known during compilation, either because the aggregate contains elements having adjustable lengths or dimensions, or because the aggregate is dynamically defined. In these cases, the word "adjustable" or "defined" appears in the "length in bytes" column.

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

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

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

STORAGE REQUIREMENTS

If the option STORAGE applies, the compiler lists the following information under the heading "Storage Requirements" on the page.

Chapter 4: The Compiler 37
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 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 contains all storage for variables declared STATIC INTERNAL.

STATEMENT OFFSET ADDRESSES

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

t+1;unc

The following method can be used to find the statement number that corresponds to an offset given in an execution-time error message.

1. From the error message, find the offset that is calculated from a procedure or ON statement.

2. In the table of offsets, locate the offsets for the named procedure or on-unit, and within this section find the largest offset that is less than the offset given in the error message. Note the corresponding statement or line number.

3. In the source listing, refer to the statement or line number. If this is not a BEGIN statement, it is the statement at which the error occurred. If it is a BEGIN statement, locate the offsets for the begin block in the table of offsets (look for the statement or line number), and find the largest offset that is less than the begin block offset given in the error message. Note the statement or line number, and repeat from (3).

EXTERNAL SYMBOL DICTIONARY

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

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

<table>
<thead>
<tr>
<th>SYMBOL</th>
<th>An 8-character field that identifies the external symbol.</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPE</td>
<td>Two characters from the following list to identify the type of entry:</td>
</tr>
<tr>
<td>SD</td>
<td>Section definition: the name of a control section within the object module.</td>
</tr>
<tr>
<td>CM</td>
<td>Common area: a type of control section that contains no data or executable instructions.</td>
</tr>
<tr>
<td>ER</td>
<td>External reference: an external symbol that is not defined in the object module.</td>
</tr>
<tr>
<td>WX</td>
<td>Weak external reference: an external symbol that is not defined in this module and that is not to be resolved unless an ER entry is encountered for the same reference.</td>
</tr>
<tr>
<td>PR</td>
<td>Pseudo-register: a field in a communications area, the task communications area (TCA), used by the compiler and by the library subroutines for handling files and controlled variables.</td>
</tr>
<tr>
<td>LD</td>
<td>Label definition: the name of an entry point to the external procedure other than that used as the name of the program control section.</td>
</tr>
<tr>
<td>ID</td>
<td>Four-digit hexadecimal number: all entries in the ESD, except LD-type entries, are numbered sequentially, commencing from 0001.</td>
</tr>
</tbody>
</table>

38  OS PL/I Optimizing Compiler: Programmer's Guide
ADDRES - Hexadecimal representation of the address of the external symbol.

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

**ESD Entries**

The external symbol dictionary always starts with the following standard entries; the entries for an external procedure with the label NAME are shown in Figure 4-9.

- **SD-type entry for PLISTART.** This control section transfers control to the initialization routine IBMPIR. When initialization is complete, control passes to the address stored in the control section PLIMAIN. (Initialization is required only once during the execution of a PL/I program, even if it calls another external procedure; in such a case, control passes directly to the entry point named in the CALL statement, and not to the address contained in PLIMAIN.)

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

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

**Other ESD Entries**

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

- **SD-type entry for the 4-byte control section PLIMAIN, which contains the address of the primary entry point to the external procedure.** This control section is present only if the procedure statement includes the option MAIN.

- **Weak external reference to PLITAES, a library subroutine that contains the standard or locally-defined tab setting for stream-oriented output.**

- **LD-type entries for all names of entry points to the external procedure.**

- **A PR-type entry for each block in compilation.**

- **ER-type entries for all the library subroutines and external procedures called by the source program.** This list includes the names of resident library subroutines called directly by compiled code (first-level subroutines), and the names of other resident library subroutines that are called by the first-level subroutines.

- **CM-type entries for non-string element variables declared STATIC EXTERNAL without the INITIAL attribute.**

- **SD-type entries for all other STATIC EXTERNAL variables and for external file names.**

- **PR-type entries for all file names.** For external file names, the name of the pseudo-register is the same as the file name; for internal file names, the compiler generates names as for the display pseudo-registers.

- **PR-type entries for all controlled variables.** For external variables, the name of the variable is used for the pseudo-register name; for internal

---

**Figure 4-9. Standard entries in the ESD**

<table>
<thead>
<tr>
<th>SYMBOL</th>
<th>TYPE</th>
<th>ID</th>
<th>ADDR</th>
<th>LENGTH</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLISTART</td>
<td>SD</td>
<td>0001</td>
<td>000000</td>
<td>000034</td>
</tr>
<tr>
<td>***NAME1</td>
<td>SD</td>
<td>0002</td>
<td>000000</td>
<td>000100</td>
</tr>
<tr>
<td>***NAME2</td>
<td>SD</td>
<td>0003</td>
<td>000000</td>
<td>000100</td>
</tr>
<tr>
<td>PLITAES</td>
<td>WX</td>
<td>0004</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td>IBMPIRA</td>
<td>ER</td>
<td>0005</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td>IBMPIRD</td>
<td>ER</td>
<td>0006</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td>IBMPIRC</td>
<td>ER</td>
<td>0007</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td>PLICALLA</td>
<td>LD</td>
<td></td>
<td>000006</td>
<td></td>
</tr>
<tr>
<td>PLICALLB</td>
<td>LD</td>
<td></td>
<td>00000A</td>
<td></td>
</tr>
<tr>
<td>PLIMAIN</td>
<td>SD</td>
<td>0008</td>
<td>000000</td>
<td>000004</td>
</tr>
</tbody>
</table>

Chapter 4: The Compiler 39
variables, the compiler generates names.

STATIC INTERNAL STORAGE MAP

If the option MAP applies, the compiler generates a listing of the contents of the static internal control section; this listing is termed the static internal storage map.

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

OBJECT LISTING

If the option LIST applies, the compiler generates a listing of the machine instructions of the object module, including any compiler-generated subroutines, in a form similar to IBM System/360 assembler language.

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

MESSAGES

If the preprocessor or the compiler detects an error, or the possibility of an error, they generate messages. Messages generated by the preprocessor appear in the listing immediately after the listing of the statements processed by the preprocessor. Messages generated by the compiler appear at the end of the listing. All messages are graded according to their severity, as follows:

• An informative (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 describes an error detected by the compiler for which the compiler has applied a "fix-up" with confidence. The resulting program will execute and will probably give correct results.

• A severe error (S) message specifies an error detected by the compiler for which the compiler cannot apply a "fix-up" with confidence. The resulting program will execute but will not give correct results.

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

The compiler lists only those messages with a severity equal to or greater than that specified by the FLAG option, as shown in Figure 4-10.

<table>
<thead>
<tr>
<th>Type of message</th>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Informatory</td>
<td>FLAG(I)</td>
</tr>
<tr>
<td>Warning</td>
<td>FLAG(W)</td>
</tr>
<tr>
<td>Error</td>
<td>FLAG(E)</td>
</tr>
<tr>
<td>Severe Error</td>
<td>FLAG(S)</td>
</tr>
<tr>
<td>Unrecoverable Error</td>
<td>Always listed</td>
</tr>
</tbody>
</table>

Figure 4-10: Selecting the lowest severity of messages to be printed, using the FLAG option

Each message is identified by a 6-character code of the form IELnmmmI, where:

• The first three characters "IEL" identify the message as coming from the optimizing compiler.

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

• The last character "I" is an operating system code for the operator indicating that the message is for information only.

The text of each message, an explanation, and any recommended programmer response, are given in the messages publication for this compiler.
RETURN CODES

For every compilation job or job step, the compiler generates a return code that indicates to the operating system the degree of success or failure it achieved. This code appears in the "end of step" message that follows the listing of the job control statements and job scheduler messages for each step. The meanings of the codes are given in Figure 4-11.

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>No error detected; compilation completed; successful execution anticipated.</td>
</tr>
<tr>
<td>0004</td>
<td>Possible error (warning) detected; compilation completed; successful execution probable.</td>
</tr>
<tr>
<td>0008</td>
<td>Error detected; compilation completed; successful execution probable.</td>
</tr>
<tr>
<td>0012</td>
<td>Severe error detected; compilation may have been completed; successful execution improbable.</td>
</tr>
<tr>
<td>0016</td>
<td>Unrecoverable error detected; compilation terminated abnormally; successful execution impossible.</td>
</tr>
</tbody>
</table>

Figure 4-11. Return codes from compilation of a PL/I program

Batched Compilation

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

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

The method of coding a PROCESS statement and the options that may be included are described under "Optional Facilities," earlier in this chapter. The options specified in a PROCESS statement apply to the compilation of the source statements between that PROCESS statement and the next PROCESS statement. Options other than these, either the defaults or those specified in the PARM field, will also apply to the compilation of these source statements. Two options, the SIZE option and the NAME option have a particular significance in batched compilations, and are discussed below.

SIZE Option

In a batched compilation, the SIZE specified in the first procedure of a batch (by a PROCESS or EXEC statement, or by default) is used throughout. If SIZE is specified in subsequent procedures of the batch, it is diagnosed and ignored. The compiler does not reorganize its storage between procedures of a batch.

NAME Option

The NAME option specifies that the compiler is to place a linkage editor NAME statement as the last statement of the object module. The use of this option in the PARM parameter of the EXEC statement, or in a PROCESS statement determines how the object modules produced by a batched compilation will be handled by the linkage editor. When the batch of object modules is link-edited, the linkage editor combines all the object modules between one NAME statement and the preceding NAME statement into a single load module; it takes the name of the load module from the NAME statement that follows the last object module that is to be included. When combining two object modules into one load module, the NAME option should not be used in the EXEC statement. An example of the use of the NAME option is given in Figure 4-12.
/* PROCESS NAME('A');
   ALPHA: PROC OPTIONS(MAIN);
   .
   .
   END ALPHA;
*/ PROCESS;
BETA: PROC;
. .
. .
END BETA;
/* PROCESS NAME('B');
GAMMA: PROC;
. .
. .
END GAMMA;

Figure 4-12. Use of the NAME option in batched compilation

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

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

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

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

Return Codes in Batched Compilation

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

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

Examples of Batched Compilations

If the external procedures are components of a language program and need to be executed together, they can be link-edited together and executed in subsequent job steps. Cataloged procedure PLIXCG can be used, as shown in Figure 4-13.

Figure 4-13. Example of batched compilation, including execution

Figure 4-14. Example of batched compilation, excluding execution
If the external procedures are independent programs to be invoked individually from a load module library, cataloged procedure PLIXCL can be used. For example, a job that contains three compile-and-link-edit operations can be run as a single batched compilation, as shown in Figure 4-14.

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

//OPTEX JOB
//JOBLIB DD DSNAME=PUBPGM,DISP=SHR
//J2 EXEC PGM=PROG2
//SYSIN DD *

Data processed by program PROG2

Compile-time Processing (preprocessing)

The preprocessing facilities of the compiler are described in the language reference manual for this compiler. You can include in a PL/I program statements that, when executed by the preprocessor stage of the compiler, modify the source program or cause additional source statements to be included from a library. The following discussion supplements the information contained in the language reference manual by providing some illustrations of the use of the preprocessor and explaining how to establish and use source statement libraries.

INVOKING THE PREPROCESSOR

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

The term MACRO owes its origin to the similarity of some applications of the preprocessor to the macro language available with such processors as the IBM System/360 Assembler. Such a macro language allows you to write a single instruction in a program to represent a sequence of instructions that have previously been defined.

The format of the preprocessor output is given in Figure 4-15.

Columns 2-72 Source program. If the original source program used more than 71 columns, then additional lines are included for any lines that need continuation. If the original source program used less than 71 columns, then extra blanks are added on the right.

Columns 73-80 Sequence number, right-aligned. If either SEQUENCE or NUMBER apply, this is taken from the sequence number field. Otherwise, it is a preprocessor generated number, in the range 1 through 99999. This sequence number will be used in the listing produced by the INSOURCE and SOURCE options, and in any preprocessor diagnostic messages.

Column 81 blank

Columns 82,83 Two-digit number giving the maximum depth of replacement by the preprocessor for this line. If no replacement occurs, the columns are blank.

Column 84 "E" signifying that an error has occurred while replacement is being attempted. If no error has occurred, the column is blank.

Figure 4-15. Format of the preprocessor output

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

A simple example of the use of the preprocessor to produce a source deck for a procedure SUBFUN is shown in Figure 4-16;
according to the value assigned to the preprocessor variable USE, the source statements will represent either a subroutine or a function.

THE %INCLUDE STATEMENT

The language reference manual for this compiler describes how to use the %INCLUDE statement to incorporate source statements from a library into a PL/I program. (A library is a type of data set that can be used for the storage of other data sets, termed members.) A set of source statements that you may wish to insert into a PL/I program by means of a %INCLUDE statement must exist as a data set (member) within a library. Creating a library and placing members in this library, are described in Chapter 10.

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

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

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

If you omit the ddbname from any pair of identifiers in a %INCLUDE statement, the preprocessor assumes the ddbname SYSLIB. In such a case, you must include a DD statement with the name SYSLIB. (The IBM-supplied cataloged procedures define a DD statement with this name in the compilation procedure step.)

The preprocessor will not recognize a PROCESS statement in a source statement module included by a %INCLUDE statement. The presence of such a PROCESS statement will result in an error in the compilation.

The use of a %INCLUDE statement to include the source statements for SUBPROC in the procedure TEST is shown in Figure 4-17. The library NEWLIB is defined in the DD statement with the qualified name PLI.SYSLIB, which is added to the statements of the cataloged procedure PLIXCLG for this job. Since the source statement library is defined by a DD statement with the name SYSLIB, the %INCLUDE statement need not include a ddbname.

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

**Dynamic Invocation of the Compiler**

You can invoke the optimizing compiler from an assembler language program by using one of the macro instructions ATTACH, CALL, LINK, or XCTL. The following information supplements the description of these macro instructions given in the manual OS/360 Supervisor and Data Management Macro Instructions.

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

You can pass three address parameters to the compiler:

1. The address of a compiler option list.

2. The address of a list of ddnames for the data sets used by the compiler.

3. The address of a page number that is to be used for the first page of the compiler listing on SYSPRINT.

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

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

The format of the three parameters for all the macro instructions is described below.

**OPTION LIST**

The option list must begin on a halfword boundary. The first two bytes contain a binary count of the number of bytes in the list (excluding the count field). The remainder of the list can comprise any of the compiler option keywords, separated by one or more blanks, a comma, or both of these.

Figure 4-17. Including source statements from a library
The ddname list must begin on a halfword boundary. The first two bytes contain a binary count of the number of bytes in the list (excluding the count field). Each entry in the list must occupy an 8-byte field; the sequence of entries is given in Figure 4-18.

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

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

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

<table>
<thead>
<tr>
<th>Entry</th>
<th>Standard ddname</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SYSLIN</td>
</tr>
<tr>
<td>2</td>
<td>not applicable</td>
</tr>
<tr>
<td>3</td>
<td>not applicable</td>
</tr>
<tr>
<td>4</td>
<td>SYSLIB</td>
</tr>
<tr>
<td>5</td>
<td>SYSIN</td>
</tr>
<tr>
<td>6</td>
<td>SYSPRINT</td>
</tr>
<tr>
<td>7</td>
<td>SYSFUNCH</td>
</tr>
<tr>
<td>8</td>
<td>SYSUT1</td>
</tr>
<tr>
<td>9</td>
<td>not applicable</td>
</tr>
<tr>
<td>10</td>
<td>not applicable</td>
</tr>
<tr>
<td>11</td>
<td>not applicable</td>
</tr>
<tr>
<td>12</td>
<td>not applicable</td>
</tr>
<tr>
<td>13</td>
<td>not applicable</td>
</tr>
<tr>
<td>14</td>
<td>SYSCIN</td>
</tr>
</tbody>
</table>

Figure 4-18. The sequence of entries in the ddname list
Chapter 5: The Linkage Editor and the Loader

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

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

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

Basic Differences

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

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

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

Choice of Program

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

Linkage Editor

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

Loader

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

The NE attribute is given to a load module that has no external symbol dictionary (ESD); a load module without an ESD cannot be processed again, either by the linkage editor or by the loader.
• The loader can accept input from its primary input source and from the automatic call library (SYSLIB).

Performance Considerations

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

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

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

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

Module Structure

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

• Text (TXT)
• External symbol dictionary (ESD)
• Relocation dictionary (RLD)

Text

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

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

External Symbol Dictionary

The external symbol dictionary (ESD) is a table containing all the external symbols that appear in the object module. An external symbol is a name that can be referred to in a control section other than the one in which it is defined.

The names of the control sections are themselves external symbols, as are the names of variables declared with the EXTERNAL attribute and entry names in the external procedure of a PL/I program. References to external symbols defined elsewhere are also considered to be external symbols; they are known as external references. Such external references in an object module always include the names of the subroutines from
The number of IDR entries that follow. This is always "1" for the optimizing compiler.

The program number of the compiler. (5734-PL1 for the optimizing compiler.)

The release number of the compiler. For example, '0102' indicates Release 1.2.

The date in day-month-year form.

Figure 5-1. The CSECT IDR information

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

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

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

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

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

Chapter 5: The Linkage Editor and the Loader
used for the PL/I transient library
subroutines, for example those concerned
with input and output (including those used
for opening and closing files), and those
that generate execution-time messages.

In basic processing, as shown in Figure
5-2, the linkage editor accepts from its
primary input source a data set defined by
the DD statement with the name SYSLIN. For
a PL/I program, this input is the object
module produced by the compiler. The
linkage editor uses the external symbol
dictionary in this object module to
determine whether the module includes any
external references for which there are no
corresponding external symbols in the
module: it attempts to resolve such
references by a method termed automatic
library call.

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

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

Any linkage editor processing additional
to the basic processing described above
must be specified by linkage editor control
statements placed in the primary input.

These control statements are described in
"Additional Processing," later in this
chapter.

Main Storage Requirements

The F-level linkage editor has three
different versions requiring differing
amounts of main storage: 44K, 88K, and
128K bytes. The F-level linkage editor is
described in the linkage editor and loader
publication.

Job Control Language for the Linkage
Editor

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

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

The name of the linkage editor is of the form IEWLFxxx, where "xxx" indicates the amount of main storage required for its execution, as shown in Figure 5-3.

<table>
<thead>
<tr>
<th>xxx</th>
<th>Amount of main storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>440</td>
<td>44K</td>
</tr>
<tr>
<td>880</td>
<td>88K</td>
</tr>
<tr>
<td>128</td>
<td>128K</td>
</tr>
</tbody>
</table>

Figure 5-3. Main storage requirements for linkage editor IEWLFxxx

The aliases IEWL or LINKEDIT are often used for the linkage editor and normally refer to the version at your installation with the largest design level. You should find out what versions are available at your installation.

The basic EXEC statement is:

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

By using the PARM parameter of the EXEC statement, you can select one or more of the optional facilities provided by the linkage editor; these facilities are described in "Optional Facilities," later in this chapter.

DD STATEMENTS FOR THE STANDARD DATA SETS

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

A fifth data set, defined by a DD statement with the name SYSLIB, is necessary if you want to use automatic library call. The five data set names, together with other characteristics of the data sets, are shown in Figure 5-4.

Primary Input (SYSLIN)

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

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

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

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

```
//IEGC JOB
//STEP1 EXEC PLIXLG
//IKED.SYSIN DD *

(insert here the object module to be link edited and executed)

/*
```

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

You can include load modules or object modules from one or more libraries in the primary input by using a linkage editor INCLUDE statement as described in "Additional Processing," later in this chapter.
Possible device classes

<table>
<thead>
<tr>
<th>ddbname</th>
<th>Contents</th>
<th>UNIT=SYSSQ or input job stream (specified by DD *)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYSLIN</td>
<td>Primary input data, normally</td>
<td>UNIT=SYSDA</td>
</tr>
<tr>
<td></td>
<td>the compiler output</td>
<td></td>
</tr>
<tr>
<td>SYSLMOD</td>
<td>Load module</td>
<td>UNIT=SYSDA</td>
</tr>
<tr>
<td>SYSUT1</td>
<td>Temporary workspace</td>
<td>UNIT=SYSDA</td>
</tr>
<tr>
<td>SYSPRINT</td>
<td>Listing, including messages</td>
<td>UNIT=SYSSQ (or SYSOUT=)</td>
</tr>
<tr>
<td>SYSLIB</td>
<td>Automatic call library</td>
<td>UNIT=SYSDA</td>
</tr>
<tr>
<td></td>
<td>(normally the PL/I resident library)</td>
<td></td>
</tr>
</tbody>
</table>

1SYSSQ Magnetic tape or direct-access device

SYSDA Direct access device

Figure 5-4. Linkage editor standard data sets

Output (SYSLMOD)

Output (that is, one or more load modules) from the linkage editor is always stored in a data set defined by the DD statement with the name SYSLMOD, unless you specify otherwise. This data set is usually called a library; libraries are fully described in Chapter 10.

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

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

This statement defines a temporary library named &&GOSST and assigns the member name GO to the load module produced by the linkage editor. To retain the load module after execution of the job, replace this DD statement with one that defines a permanent library. For example, assume that you have a permanent library called USLIB on 2311 disk pack serial number 371; to name the load module MOD1 and place it in this library, code:

```
//LKED.SYSLMOD DD DSNAME=USLIB(MOD1),
// UNIT=2311,VCL=SER=371,DISP=CLD
```

The size of a load module must not exceed 512K bytes for programs executed under MFT; a much larger load module is allowed for MVT. The SPACE parameter in the DD statement with the name SYSLMOD used in the IBM-supplied cataloged procedures allows for an initial allocation of 50K bytes and, if necessary, 15 further allocations of 20K bytes (a total of 350K bytes); this should suffice for most applications.

Temporary Workspace (SYSUT1)

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

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

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

If your installation supports dedicated workfiles, these can be used to provide temporary workspace for the link-edit job step, as described in Chapter 11.

Automatic Call Library (SYSLIB)

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

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

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

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

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

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

**Listing (SYSPRINT)**

The linkage editor generates a listing that includes reference tables relating to the load modules that it produces and also, when necessary, messages. The information that may appear is described under "Listing Produced by the Linkage Editor," later in this chapter.

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

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

**EXAMPLE OF LINKAGE EDITOR JCL**

A typical sequence of job control statements for link editing an object module is shown in Figure 5-5. The DD statement SYSLIN indicates that the object module will follow immediately in the input stream; for example, it might be an object deck created by invoking the optimizing compiler with the DECK option, as described in Chapter 4. The DD statement with the name SYSLMOD specifies that the linkage editor is to name the load module LKEX, and that it is to place it in a new library named MODLIB; the keyword NEW in the DISP parameter indicates to the operating system that this DD statement specifies the creation of a library.

**Optional Facilities**

The linkage editor provides a number of optional facilities that are selected by including the appropriate keywords from the following list in the PARM parameter of the EXEC statement that invokes it:

- LIST
- MAP or XREF
- LET or XCALL
- NCAL
- SIZE

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

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

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

```
//STEPA EXEC PLIXCLG,PARM=EXE='LIST,XREF'
```

The linkage editor options are of two types:

1. Simple keywords, for example, LIST, that specifies a facility. LET, LIST, MAP, NCAL, XCALL, and XREF are of this type.
Figure 5-5. Typical job control statements for link editing a PL/I program

2. Keywords that permit you to assign a value to a function (for example, SIZE=10K).

The linkage editor options are described in the following sections, in alphabetic order.

**LET Option**

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

**LIST Option**

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

**MAP Option**

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

**NCAL Option**

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

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

**SIZE Option**

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

\[
\text{SIZE}=(m, n)
\]

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

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

Figure 5-6 gives values for "m" and "n" for the three versions of the F-level linkage editor.

54 OS PL/I Optimizing Compiler: Programmer's Guide
If you specify SIZE incorrectly, or if you omit it, default values set at system generation are used. If you specify SIZE greater than the region or partition size, the maximum amount of main storage will be used.

**XCAL Option**

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

**XREF Option**

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

**Listing Produced by the Linkage Editor**

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

The listing comprises a small amount of standard information that always appears, together with those items of optional information specified in the PARM parameter of the EXEC statement that invokes the linkage editor, or that are applied by default. The optional components of the listing and the corresponding linkage editor options are as shown in Figure 5-7.

<table>
<thead>
<tr>
<th>Listings</th>
<th>Options Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control statements processed by the linkage editor</td>
<td>LIST</td>
</tr>
<tr>
<td>Map of the load module</td>
<td>MAP or XREF</td>
</tr>
<tr>
<td>Cross-reference table</td>
<td>XREF</td>
</tr>
</tbody>
</table>

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

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

An example of the listing produced for a typical PL/I program is given in Appendix F.

**Diagnostic Messages and Control Statements**

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

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

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

- The first three characters "IEW" identify the message as coming from the linkage editor.
- The next three characters are a 3-digit
The last character "x" is a severity code. The possible severity codes and their meanings are given in Figure 5-8.

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

The warning message Ipw0461, together with a return code of 0004, frequently appears in the linkage editor listing for a PL/I program. It refers to external references that have not been resolved because NCAL is specified. The references occur in PL/I library subroutines that are link edited with your program as a result of automatic library call. Some library subroutines may, in turn, call other library subroutines. For these secondary subroutines that are required, the compiler generates another external symbol dictionary containing alternative names for the subroutines. These new references can be resolved, and the required subroutines placed in the new load module. If the secondary subroutines in turn call other subroutines, the process is repeated.

### Module Map

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

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

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

**Module Map**

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

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

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

**Diagnostic Message Directory**

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

The warning message Ipw0461, together with a return code of 0004, frequently appears in the linkage editor listing for a PL/I program. It refers to external references that have not been resolved because NCAL is specified. The references occur in PL/I library subroutines that are link edited with your program as a result of automatic library call. Some library subroutines may, in turn, call other library subroutines. For these secondary subroutines that are required, the compiler generates another external symbol dictionary containing alternative names for the subroutines. These new references can be resolved, and the required subroutines placed in the new load module. If the secondary subroutines in turn call other subroutines, the process is repeated.

Figure 5-8. Diagnostic message severity codes

**Figure 5-8. Diagnostic message severity codes**

At the end of the listing, immediately preceding the "Diagnostic Message Directory" (described later in this chapter), the linkage editor places a statement of the disposition of the load module. The disposition statements, with one exception, are self-explanatory; the exception is:

**###module name DOES NOT EXIST BUT HAS BEEN ADDED TO DATA SET**

This appears when the NAME statement has been used to add a new module to the data set defined by the DD statement with the name SYSMOD. The use of the NAME statement is described under "Module Name," later in this chapter. If you name a new module by including its name in the DSNAMe parameter of the DD statement with the name SYSMOD, the linkage editor assumes that you want to replace an existing module (even if the data set is new).
Figure 5-9. Return codes from the linkage editor

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

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

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

Return Code

For every linkage editor job or job step, the linkage editor generates a return code that indicates to the operating system the degree of success or failure it achieved. This code appears in the "end of step" message and is derived by multiplying the highest severity code (see "Diagnostic Messages and Control Statements," earlier in this chapter) by four, as shown in Figure 5-9.

The return code 0004 almost invariably appears after a PL/I program has been link edited because some external references will not have been resolved. (Refer to "Diagnostic Message Directory," earlier in this chapter.)

Additional Processing

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

CROSS-REFERENCE TABLE

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

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

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

FORMAT OF CONTROL STATEMENTS

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

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

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

MODULE NAME

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

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

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

2. If you are producing two or more load modules from a single link-edit job step, you will need to use the NAME statement. (The optimizing compiler can supply the NAME statements when you use batch compilation as described in Chapter 4.)

The format of the NAME statement is:

```
NAME name{(R)}
```

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

- It identifies a load module. The name specified will be given to the load module. "(R)", if present, specifies that the load module is to replace an existing load module of the same name in the data set defined by the DD statement with the name SYSLMOD.

- It acts as a delimiter between input for different load modules in one link-edit step.

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

Alternative Names

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

The format of the ALIAS statement is:

```
ALIAS name
```

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

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

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

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

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

ADDITIONAL INPUT SOURCES

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

INCLUDE Statement

The INCLUDE statement causes the linkage editor to process the module or modules indicated. After the included modules have been processed, the linkage editor continues with the next item in the primary input. If an included sequential data set also contains an INCLUDE statement, that statement is processed as if it were the last item in the data set, as shown in Figure 5-10.

```
<table>
<thead>
<tr>
<th>Primary Input Sequential Library</th>
<th>Data Set</th>
<th>Data Set</th>
<th>Member</th>
</tr>
</thead>
<tbody>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>end</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>INCLUDE</td>
<td>end</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>---</td>
<td>INCLUDE</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>not</td>
<td>---</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>processed</td>
<td>---</td>
</tr>
<tr>
<td>end</td>
<td>end</td>
<td>end</td>
<td>end</td>
</tr>
</tbody>
</table>
```

Figure 5-10. Processing of additional data sources

The format of the INCLUDE statement is:

```
INCLUDE ddname((membername))
```

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

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

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

LIBRARY Statement

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

The format of the LIBRARY statement is:

```
LIBRARY ddname(membername)
```

OVERLAY STRUCTURES

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

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

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

Before preparing the linkage editor control statements, you must design the overlay structure for your program. A tree is a graphic representation of an overlay structure that shows which segments occupy main storage at different times. The design of trees is discussed in the linkage editor and loader publication, but for the purposes of this chapter, Figure 5-11 contains a simple example. The program comprises six procedures: A, B, C, D, E.

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

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

The most efficient method of defining an overlay structure, and the simplest for a PL/I program, is to group all the OVERLAY and INSERT statements together and place them in the linkage editor input (SYSLIN) storage; thus the lines representing procedures D and E start from the node X.

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

Control Statements

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

The format of the OVERLAY statement is:

OVERLAY symbol

where "symbol" is the node at which the segment starts (for example, X in Figure 5-11). You must specify the start of every segment, except the root segment, in an OVERLAY statement.

The format of the INSERT statement is:

INSERT control-section-name

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

Figure 5-12. Creating and executing the overlay structure of Figure 5-11

and F. Procedure B calls procedure C which, in turn, calls procedures D and E. (Only procedure A requires the option MAIN.)

The main procedure (A) must be in main storage throughout the execution of the program. Since the execution of procedure B will be completed before procedure F is called, the two procedures can occupy the same storage; this is depicted by the lines representing the two procedures in Figure 5-11 starting from the common point (node) X. Procedure B must remain in storage while procedures C, D, and E are executed, but procedures D and E can occupy the same
after the object modules that form the program. The linkage editor initially places all these object modules in the root segment, and then moves those control sections that are referred to in INSERT statements into other segments.

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

The use of the IBM-supplied cataloged procedure PLIXCLG to create and execute the overlay structure of Figure 5-11, is shown in Figure 5-12.

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

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

It is important that the PL/I library subroutines be in the root segment, since the optimizing compiler does not support exclusive calls (calls between segments that do not lie in the same path). For example, in Figure 5-11, procedures in the segment containing D could call procedures in the segments containing A, B, C, and D, but not in the segments containing E or F. Procedures in the segments containing B or C could call procedures in the segments containing A, B, C, D, and E, but not in the segment containing F. A procedure in the segment containing B may not call a procedure in the segment containing A if this latter procedure calls a procedure in the segment containing F.

However, certain library subroutines may not be required by all segments, in which case you can move them into a lower segment. To do this, compile the procedures using the compiler option ESD, and examine the resulting external symbol dictionary. For example, if in Figure 5-11 a library subroutine is called only by the segment containing E, you can move it into that segment by placing an INSERT statement, specifying the subroutine name, immediately after the statement INSERT ******F1.

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

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

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

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

**LINK EDITING FETCHABLE LOAD MODULES**

The PL/I FETCH and RELEASE statements permit the dynamic loading of separate load modules which can be subsequently invoked from the PL/I object program.

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

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

- An ENTRY statement to define the entry-point into the PL/I program.
Control Section Present in
PLISTART All programs
IBMBJWT1 Programs that use the WAIT statement
IBMTJWT1 Multitasking programs that use the WAIT statement
IBMBTOC1 Programs that use the COMPLETION built-in function or pseudovariable
IBMTTOC1 Multitasking programs that use the COMPLETION built-in function or pseudovariable.
IBMTPR1 Programs that use the PRIORITY pseudovariable
IBMTTPR1 Multitasking programs that use the PRIORITY pseudovariable.
IBMSEFL1 Programs compiled with the FLOW or COUNT options.

Figure 5-13. Control sections to be deleted for optimum space-saving

//FETCH JOB
//STP EXEC PLIXCL
//PLI.SYSIN DD *
.
.
 PL/I source(fetchable)
.
.
/*
//LKED.SYSLMOD DD DSN=PRVLIB,...
//LKED.SYSLIN DD DDNAME=SYSIN
// DD DSN=&&LOADSET,DISP=(OLD,DELETE)
//LKED.SYSIN DD *
ENTRY procedure-name
REPLACE PLISTART
REPLACE IBMBJWT1
REPLACE IBMBTOC1
REPLACE IBMTPR1
NAME FETCH1
/*

Figure 5-14. Example of link-editing a fetchable load module

• A NAME statement to define the name used for the fetchable load module. This statement is required if the compiler option NAME is not used and if the name is not specified in the DSN parameter in the SYSLMOD DD statement used to define the load module library.

• Optionally, for optimum space saving, REPLACE statements to delete the control sections shown in Figure 5-13, if they are present in the object module.

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

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

The job control statements and the linkage editor statements to link-edit a fetchable load module into a library called PRVLIB are given in Figure 5-14. The cataloged procedure PLIXCL is used to illustrate these statements by sharing a job that includes both the compilation and the link-editing of the fetchable PL/I module.

Combining PL/I Modules from the Optimizing and Checkout Compilers

When a program is to consist of PL/I modules compiled by the optimizing and checkout compilers, the following points should be considered before link-editing the modules into a single load module:

• The modules compiled by the optimizing compiler should be link-edited to form a load module.

• The linkage editor option NCAL must be specified for this link-editing operation.

• The load module containing the modules compiled by the optimizing compiler can now be link-edited with the link-edit stubs produced by the checkout compiler. This method ensures that the initialization routine for a program compiled by the optimizing compiler will not be included in the final load module and that the initialization routine for a program compiled by the checkout compiler is used when the program is executed.

Both the space occupied by the final load module and its speed of execution are affected by the SYSLIB data set specified for use by the linkage editor. Two data sets, SYS1.PLICMIX and SYS1.PLIBASE are available. Use SYS1.PLICMIX to obtain a
smaller load module at the expense of execution time; use SYSl.PLIBASE to save execution time at the expense of space.

Loader

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

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

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

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

Main Storage Requirements

The minimum main storage requirements for the loader are shown in Figure 5-17.

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

Job Control Language for the Loader

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

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

- PLIXCG Compile, load-and-execute
- PLIXG Load-and-execute

The following paragraphs describe the essential JCL statements for the loader. The IBM-supplied cataloged procedures are described in Chapter 11 and include examples of these statements.
Figure 5-15. Basic loader processing

Figure 5-16. Loader processing, link-pack area and SYSLIB resolution
EXEC STATEMENT

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

```
//stepname EXEC PGM=LOADER
```

By using the PARM parameter of the EXEC statement, you can select one or more of the optional facilities provided by the loader; these are described under "Optional Facilities," later in this chapter.

DO STATEMENTS FOR THE STANDARD DATA SETS

The loader always requires one standard data set; that defined by the DD statement

with the name SYSLIN. Three other standard data sets are optional and if you use them you must define them in DD statements with the names SYSLCUT, SYSPRINT, and SYSLIB. The four data sets, their names, and other characteristics of the data sets, are shown in Figure 5-18.

The data sets defined by the DD statements with the names SYSLIN, SYSLIB, and SYSLCUT are those specified at system generation for your installation. Other ddnames may have been specified at your installation; if they have, your JCL statements must use them in place of those given above. In a similar manner the IBM-supplied cataloged procedures PLIXCG and PLIXG use names as shown above; your systems programmer will have to modify these procedures if the names at your installation are different.

Primary Input (SYSLIN)

Primary input to the loader must be a standard data set defined by a DD statement with the name SYSLIN; this data set must have consecutive organization. The input can comprise one or more object modules, one or more load modules, or a mixture of object modules and load modules.

For a PL/I program the primary input is usually a data set containing an object module produced by the compiler. This data set may be on magnetic tape or a direct-access device, or you can include it in the input job stream. In all cases the input must be in the form of 80-byte F-format records.

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

//SYSLIN DD DSN=&&LOADSET,DISP=(OLD,DELETE)

This statement specifies that the data set &&LOADSET is temporary. If you want to modify this statement you must refer to it by the qualified ddbname GO.SYSLIN.

The IBM-supplied cataloged procedure PLIXG does not include a DD statement for the input data set; you must always supply one, specifying the characteristics of your input data set using the qualified ddbname GO.SYSLIN.

Automatic Call Library (SYSLIB)

Unless you specify otherwise, the loader will normally attempt to resolve external references by automatic library call. The automatic call library (SYSLIB), and how to specify it, is described in the linkage editor section earlier in this chapter.

Listing (SYSLOUT)

The loader generates a listing that includes a module map (if you have specified the MAP option) and, if errors have been detected during processing, messages referring to these. The information that may appear is described in "Listing Produced by the Loader," later in this chapter.

You must define the data set in which you want this listing to be stored by a DD statement with the name SYSLUT and it must have consecutive organization. Although the listing is usually printed it can be stored on any magnetic-tape or direct-access device. For printed output the following DD statement will suffice:

//SYSLOUT DD SYSOUT=A

Listing (SYSPRINT)

As well as the information listed in the data set defined by the DD statement with the name SYSLOUT certain information produced by the loader is always stored in the data set defined by the DD statement with the name SYSPRINT. This data set, which must have consecutive organization, holds messages that refer to errors that have occurred during execution of your program, as well as the results produced by your program. The information that may appear is described in "Listing Produced by the Loader," later in this chapter. For printed output the following DD statement will suffice:

//SYSPRINT DD SYSOUT=A

EXAMPLES OF LOADER JCL

A sequence of job control language for the loader is shown in Figure 5-19. A PL/I program has been compiled in a job step with the step name PII; the resultant object module has been placed in the data set defined by the DD statement with the name SYSLIB. Because this module is to be loaded and executed in the same job as the compile step, this DD statement can use the backward reference, indicated by the asterisk, as shown. If the compile and load-and-go steps were in different jobs, the DD statement would have to specify a permanent data set, cataloged or uncataloged.

The IBM-supplied cataloged procedure PLIXCG includes a DD statement with the name SYSLIB in both the compile and load-and-go procedure steps; you do not need to specify this statement unless you want to modify it. The IBM-supplied cataloged procedure PLIXG does not include a DD statement with the name SYSLIB; you must supply one, using the qualified name GO.SYSLIB.

Typical job control language statements for the loader are shown in Figure 5-20. The example illustrates how to include, in the input stream, both an object module for input to the loader, and data to be used by your program during execution.

The DD statement with the name SYSLIB and the two following unnamed DD statements define three data sets that are to be concatenated into one data set to be used as input to the loader. The first data set is named OBJMOD and contains an object module. This data set could be the output of an optimizing compiler that has just processed your PL/I program. The second data set is named MODLIB(MOD55) containing a load module that has been given the name MOD55 and stored in the library called MODLIB. The third data set is an object module defined by the DD statement with the name IN. This DD statement appears further on and has the asterisk notation that indicates that the data set defined by this statement follows in the input stream.

The DD statement with the name SYSLIB and the unnamed DD statement immediately
Figure 5-19. Job control language for load-and-go

Figure 5-20. Object and load modules in load-and-go

following it define two data sets that are to be concatenated so that they can be searched for unresolved external references by automatic library call. The first data set is the PL/I resident library (SYS1.PLILIB) and the second is a private library called PRIVLIB.

Optional Facilities of the Loader

The loader provides a number of optional facilities that are selected by including the appropriate keywords from the following list in the PARM parameter of the EXEC statement that invokes it:

CALL
EF
LET
MAP
PRINT
RES
SIZE

Code the PARM parameter as follows:

PARM = '[(optionlist)/pgmparm]' where "option list" is a list of loader options, and "pgmparm" is a parameter to be passed to the main procedure of the PL/I compiler.
program to be executed. In the examples
given below, the program parameter is
referred to as PP.

If both loader options and a program
parameter occur in the PARM parameter, the
loader options are given first and are
separated from the program parameter by a
slash. If there are loader options but no
program parameter, the slash is omitted,
but if there are only program parameters
the slash must be coded. If there is more
than one option, the option keywords are
separated by commas.

The PARM field can have one of three
formats:

1. If the special characters / cr = are
   used, the field must be enclosed in
   single quotes. For example:

   PARM='MAP,EP=FIRST/PP'
   =MAP,EP=FIRST'
   PARM='/PP'

2. If these characters are not included,
   and there is more than one loader
   option, the options must be enclosed
   in parentheses. For example:

   PARM=(MAP,LET)

3. If these characters are not included,
   and there is only one loader option,
   neither quotes nor parentheses are
   required. For example:

   PARM=MAP

To overwrite the PARM parameter options
specified in a cataloged procedure, you
must refer to the PARM parameter by the
qualified name PARM.procstepname. For
example: PARM.GO.

The loader options are of two types:

1. Simple pairs of keywords: a positive
   form (for example, CALL) that requests
   a facility, and an alternative
   negative form (for example NOCALL)
   that rejects that facility. CALL,
   LET, MAP, PRINT, and RES are of this
type.

2. Keywords that permit you to assign a
   value to a function (for example,
   SIZE=10K). EP and SIZE are of this
type.

The loader options are described in the
following sections, in alphabetic order.

CALL Option

The CALL option specifies that the loader
will attempt to resolve external references
by automatic library call. To preserve
compatibility with the linkage editor, the
negative form of this option can be
specified as NCAL as well as by NOCALL.

EP Option

The EP option specifies the entry point
name of the program that is to be executed.
The format of the EP option is:

EP=name

where "name" is an external name. If all
input modules are load modules you must
specify EP=FIRSTSTART.

LET Option

The LET option specifies that the loader
will try to execute the problem program
even if a severity 2 error has been found.

MAP Option

The MAP option specifies that the loader
is to print a map of the load module giving
the relative locations and lengths of
control sections in the module. You must
specify the data set defined by the DD
statement with the name SYSLCUT to have
this map printed. The module map is
described in "Listing Produced by the
Loader," later in this chapter.

PRINT Option

The PRINT option specifies that the data
set defined by the DD statement with the
name SYSLJUT is to be used for messages,
the module map, and other loader
information.

RES Option

The RES option specifies that the loader
will attempt to resolve external references
by a search of the link-pack area of main
storage. This search will be made after
the primary input to the loader has been
processed but before the data set defined
by the DD statement with the name SYSLIB is
opened.

SIZE Option

The SIZE option specifies the amount of
main storage, in bytes, to be allocated to
the loader. The format of the SIZE option is:

SIZE=yyyyyy specifies that yyyyyy bytes of
main storage are to be
allocated to the loader.

SIZE=yyyyK specifies that yyyK bytes of
main storage are to be
allocated to the loader
(1k=1024).

The values can be enclosed, optionally,
in parentheses.

Listing Produced by the Loader

The loader can provide a listing on the
SYSLOUT data set; the SYSPRINT data set is
used by the problem program. The contents
of each is given in Figure 5-21.

Data set Contents

SYSLOUT Loader explanatory messages and
diagnostic messages, and
optionally, a module map.

SYSPRINT PL/I execution-time messages,
and program program output.

Figure 5-21. Contents of SYSLOUT and
SYSPRINT data sets

The SYSLOUT listing is described here;
the SYSPRINT listing is described in
Chapter 4.

The items in the SYSLOUT listing appear
in the following sequence:

1. Statement identifying the loader.

2. Module map (if specified).

3. Explanatory, error, or warning
   messages.

4. Diagnostic messages.

MODULE MAP

If the MAP option is specified, a module
map is printed in the SYSLOUT listing. The
map lists all the control sections in the
load module and all the entry point names
(other than the first) in each control
section. The information for each reference is:

- The control section or entry point name.
- An asterisk, if the control section is
  included by automatic library call.
- An identifier, as follows:
  
  SD Section definition: the name of
  the control section.
  
  LR Label reference: identifying an
  entry point in the control section
  other than the primary entry
  point.
  
  CM Common area: an external file, or
  a non-string element variable
  declared STATIC EXTERNAL without
  the INITIAL attribute.

- Absolute address of the control section
  or entry point.

Each reference is printed left to right
across the page and starts at a preset
position. This gives the impression that
the references are arranged in columns, but
the correct way to read the map is line by
line, across the page, not down each
column.

The module map is followed by a similar
listing of the pseudo-registers. The
identifier used here is PR, and the address
is the offset from the beginning of the
pseudo-register vector (PRV). The total
length of the PRV is given at the end.

The total length of the module to be
executed, and the absolute address of its
primary entry point, are given after the
explanatory messages and before the
diagnostic messages.

EXPLANATORY AND DIAGNOSTIC MESSAGES

The loader generates messages describing
errors or conditions, detected during
processing by the loader, that may lead to
error. The format of these messages is
given in "Diagnostic Messages and Control
Statements" in the linkage editor section
earlier in this chapter.
When the module to be executed has been processed, the loader prints out in full all the messages referred to above. The text of each message, an explanation, and any recommended programmer response, are given in the linkage editor and loader publication.

The warning message IEW1001 almost always appears in the listing. The explanation for this is the same as that for IEW0461, described under "Diagnostic Message Directory," in the linkage editor section earlier in this chapter.
This chapter describes briefly the nature and organization of data sets, the data management services provided by the operating system, and the record formats acceptable for auxiliary storage devices. The way in which a data set is associated with a PL/I file is fully described in the language reference manual for this compiler. Methods of creating and accessing data sets in PL/I are given in Chapters 7 and 8.

Data Sets

In IBM System/360 Operating System, 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 paper tape; or it may be recorded on a direct-access device such as a magnetic disk or drum. A printed listing produced by a program is a data set, but it cannot be accessed by a program.

A data set resides on one or more volumes. A volume is a standard 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 punched-card or paper-tape volume or a magnetic-tape volume either without labels or with nonstandard labels).

A magnetic-tape or direct-access volume can contain more than one data set; conversely, a single data set can span two or more magnetic-tape or direct-access volumes.

DATA SET NAMES

A data set on a direct-access device must have a name so that the operating system can refer to it. If you do not supply a name, the operating system will supply a temporary one. A data set on a magnetic-tape device must have a name if the tape has standard labels (see "Labels," later in this chapter.) A name consists of up to eight characters, the first of which must be alphabetic. Data sets on punched cards, paper tape, unlabeled magnetic tape, or nonstandard unlabeled magnetic tape do not have names.

You can place the name of a data set, with information identifying the volume on which it resides, in a catalog that exists on the volume containing the operating system. Such a data set is termed a cataloged data set. To catalog a data set use the CATLG subparameter of the DISP parameter of the DD statement. To retrieve a cataloged data set, you do not need to give the volume serial number or identify the type of device; you need only specify the name of the data set and its disposition. The operating system searches the catalog for information associated with the name and uses this information to request the operator to mount the volume containing your data set.

If you have a set of related data sets, you can increase the efficiency of the search for a particular data set by establishing a hierarchy of indexes in the catalog. For example, consider an installation that groups its data sets under four headings: ENGRNG, SCIENCE, ACCNTS, and INVMTRY, as shown in Figure 6-1. In turn, each of these groups is subdivided; for example, the SCIENCE group has subgroups called PHYSICS, CHEM, MATH, and BIOLOGY. The MATH subgroup itself contains three subgroups: ALGEBRA, CALCULUS, and BOOL.

Figure 6-1. A hierarchy of indexes
To find the data set BOOL, the names of all the indexes of which it is part must be specified, beginning with the largest group SCIENCE, followed by the subgroup name MATH and finally the data set name BOOL. The names are separated by periods. The complete identification needed to find the data set BOOL is

SCIENCE.MATH.BOOL

Such an identifier is termed a qualified name. The maximum length of a qualified name is 44 characters, including the separating periods; each component name has a maximum length of eight characters. (Do not use names that begin with the letters SYS; if the name is qualified do not use P as the nineteenth character. The names assigned by the operating system to unnamed temporary data sets are of this form, with P as the nineteenth character, and these data sets are deleted when the utility program IEMPROGM is used with a SCRATCH statement that includes the keywords VTOC and SYs.)

Some data sets are updated periodically, or are logically part of a group of data sets, each of which is related to the other in time. You can relate such data sets to each other in what is termed a generation data group. Each data set in a generation data group has the same name qualified by a unique parenthesized generation number (for example, STOCK(0), STOCK(-1), STOCK(-2)). The most recently cataloged data set is generation 0, and the preceding generations are -1, -2, and so on. You specify the number of generations to be saved when you establish the generation data group.

For example, consider a generation data group that contains a series of data sets used for weather reporting and forecasting; the name of the data sets is WEATHER. The generations for the group (assuming that three generations are to be saved) are:

WEATHER(0)
WEATHER(-1)
WEATHER(-2)

When WEATHER is updated, the new data set is specified to the operating system as WEATHER(+1). When it catalogs the new data set, the operating system changes the name to WEATHER(0), changes the former WEATHER(0) to WEATHER(-1), the former WEATHER(-1) to WEATHER(-2), and deletes the former WEATHER(-2).

To find out how to create a generation data group, refer to the job control language and utilities publications.

BLOCKS AND RECORDS

The items of data in a data set are arranged in blocks separated by interblock gaps (IBG)\(^1\).

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 code (ASCII or EBCDIC) in which the data is recorded. These codes are discussed in "Data Codes," below. Specify the block size in the BLKSIZE parameter of the DD statement or in the BLKSIZE option of the ENVIRONMENT attribute.

A record is the unit of data transmitted to and from a program. When writing a PL/I program, you need consider only the records that you are reading or writing; but when you describe the data sets that your program will create or access, you must be aware of the relationship between blocks and records.

If a block contains two or more records, the records are said to be blocked. Blocking conserves storage space in a volume because it reduces the number of 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 LRECL parameter of the DD statement or in the RECSIZE option of the ENVIRONMENT attribute.

Data Codes: The normal code in which data is recorded in System/360 is the Extended Binary Coded Decimal Interchange Code (EBCDIC) although source input can optionally be coded in BCD (Binary Coded Decimal). However, for magnetic tape only, System/360 will accept 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. For a full description of the options used for ASCII data sets see the language reference manual for this compiler.

---

\(^1\) Although the term "interrecord gap" is widely used in operating system manuals, it is not used here; it has been replaced by the more accurate term "interblock gap."
RECORD FORMATS

The records in a data set must have one of the following formats:

- **F** fixed length
- **V** variable length (D- or V-format)
- **U** undefined length

All formats can be blocked if required, but only fixed-length and variable-length records are deblocked automatically by the system; undefined length records must be deblocked by your program.

Fixed-length Records (F-format Records)

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

- **F** Fixed-length, unblocked
- **FB** Fixed-length, blocked
- **FS** Fixed-length, unblocked, standard
- **FBS** Fixed-length, blocked, standard

In a data set with fixed-length records, as shown in Figure 6-2, all records have the same length. If the records are blocked, each block contains an equal number of fixed-length records (although the last block may be truncated if there are insufficient records to fill it). If the records are unblocked, each record constitutes a block.

Unblocked records (F-format):

```plaintext
[Record] IBG [Record] IBG [Record] IBG ...
```

Blocked records (FB-format):

```plaintext
-----------Block-----------
[Record Record Record] IBG [Record]
```

Figure 6-2. Fixed-length records

Because it can base blocking and deblocking on a constant record length, the operating system can process fixed-length records faster than variable-length records. The use of "standard" (FS-format and FBS-format) records further optimizes the sequential processing of a data set on a direct-access device. A standard format data set must contain fixed-length records and must have no embedded empty tracks or short blocks (apart from the last block). With a standard format data set, the operating system can predict whether the next block of data will be on a new track and, if necessary, can select a new read/write head in anticipation of the transmission of that block. A PL/I program never places embedded short blocks in a data set with fixed-length records. A data set containing fixed-length records can be processed as a standard data set even if it is not created as such, providing it contains no embedded short blocks or empty tracks.

Variable-length Records (D- or V-format Records)

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

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

V-format permits both variable-length records and variable-length blocks. The first four 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 backwards. Illustrations of variable-length records are shown in Figure 6-3.

V-format signifies unblocked variable-length records. Each record is treated as a block containing only one record, the first four bytes of the block contain block control information, and the next four contain record control information.

VB-format signifies blocked variable-length records. Each block contains as many complete records as it can accommodate. The first four bytes of the block contain block control information, and the first four bytes of each record contain record control information.

Spanned Records: A spanned record is a variable-length record in which the length of the record can exceed the size of a block. If this occurs, the record is
V-format:

\[
\begin{array}{c|c|c}
C1 & C2 & IBG \\
\end{array}
\begin{array}{c|c|c}
\text{Record 1} & \text{IBG} & \text{Record 1} \\
\end{array}
\begin{array}{c|c|c}
\text{Record 2} & \text{IBG} & \text{Record 2} \\
\end{array}
\]

VB-format:

\[
\begin{array}{c|c|c|c}
C1 & C2 & C2 & IBG \\
\end{array}
\begin{array}{c|c|c|c}
\text{Record 1} & \text{IBG} & \text{Record 2} & \text{IBG} \\
\end{array}
\begin{array}{c|c|c|c}
\text{Record 2} & \text{IBG} & \text{Record 3} & \text{IBG} \\
\end{array}
\]

VS-format:

\[
\begin{array}{c|c|c|c|c|c|c|c|c}
C1 & C2 & C2 & IBG & C1 & C2 & C2 & IBG & C1 & C2 & IBG \\
\end{array}
\begin{array}{c|c|c|c|c|c|c|c|c}
\text{Record 1} & \text{IBG} & \text{Record 2} & \text{IBG} & \text{Record 2} & \text{IBG} & \text{Record 2} & \text{IBG} \\
\end{array}
\begin{array}{c|c|c|c|c|c|c|c|c}
\text{IBG} & \text{IBG} & \text{IBG} & \text{IBG} & \text{IBG} & \text{IBG} & \text{IBG} & \text{IBG} & \text{IBG} \\
\end{array}
\]

VBS-format:

\[
\begin{array}{c|c|c|c|c|c|c|c|c}
C1 & C2 & C2 & IBG & C1 & C2 & C2 & IBG & C1 & C2 & IBG \\
\end{array}
\begin{array}{c|c|c|c|c|c|c|c|c}
\text{Record 1} & \text{IBG} & \text{Record 2} & \text{IBG} & \text{Record 2} & \text{IBG} & \text{Record 2} & \text{IBG} \\
\end{array}
\begin{array}{c|c|c|c|c|c|c|c|c}
\text{IBG} & \text{IBG} & \text{IBG} & \text{IBG} & \text{IBG} & \text{IBG} & \text{IBG} & \text{IBG} & \text{IBG} \\
\end{array}
\]

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

Figure 6-3. Variable-length records

Divided into segments and accommodated in two or more consecutive blocks by specifying the record format as either VS or VBS. Segmentation and reassembly are handled automatically. The use of spanned records allows you to select a block size, independently of record length, that will combine optimum use of auxiliary storage with maximum efficiency of transmission.

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

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

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

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

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

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

Undefined-length Records (U-format 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 five types of data set, which differ in the way data is stored within them and in the permitted means of access to the data. The three main types of data set and the corresponding keywords describing their PL/I organization are given in Figure 6-4.

<table>
<thead>
<tr>
<th>Type of Data Set</th>
<th>PL/I Organization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>CONSECUTIVE</td>
</tr>
<tr>
<td>Indexed sequential</td>
<td>INDEXED</td>
</tr>
<tr>
<td>Direct</td>
<td>REGIONAL</td>
</tr>
</tbody>
</table>

Figure 6-4. The three main types of data set

The fourth type, teleprocessing, is recognized by the compiler by the file attribute TRANSIENT.

The fifth type, partitioned, has no corresponding PL/I organization.

A direct (or REGIONAL) data set must reside on a direct-access volume. The records within the data set can be organized in three ways: REGIONAL(1), REGIONAL(2), and REGIONAL(3); in each case, the data set is divided into regions, each of which contains one or more records. A key that specifies the region number and, for REGIONAL(2) and REGIONAL(3), identifies the record, permits direct access to any record; sequential processing is also possible.

A teleprocessing data set (associated with a TRANSIENT file in a PL/I program) must reside in main storage. Records are placed in physical sequence; a key embedded in the record provides direct access to any record.

In a partitioned data set, independent groups of sequentially organized data, each called a member, reside on a direct-access volume. The data set includes a directory that lists the location of each member. Partitioned data sets are often called libraries. The compiler includes no special facilities for creating and accessing partitioned data sets; however, this is not necessary since each member can be processed as a CONSECUTIVE data set by a PL/I program, and there is ready access to the operating system facilities for partitioned data sets through job control language. The use of partitioned data sets as libraries is described in Chapter 10.

LABELS

The operating system uses labels to identify magnetic-tape and direct-access volumes and the data sets they contain, and to store data set attributes (for example, record length and block size). The attribute information must originally come from a DD statement or from your program. Once the label is written you need not specify the information again.

Magnetic-tape volumes can have standard or nonstandard labels, or they can be unlabeled. 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 backwards.

Direct-access volumes have standard

1 Do not confuse the terms "sequential" and "direct" with the PL/I file attributes SEQUENTIAL and DIRECT. The attributes refer to how the file is to be processed, and not to the way the corresponding data set is organized.
labels. Each volume is identified by a volume label, which is stored at a standard location on the volume. This label contains a volume serial number and the address of a volume table of contents (VTOC). The table of contents, in turn, contains a label, termed a data set control block (DSCB), for each data set stored on the volume.

DATA DEFINITION (DD) STATEMENT

A data definition (DD) statement is a job control statement that defines a data set to the operating system, and is a request to the operating system for the allocation of input/output resources. Each job step must include a DD statement for each data set that is processed by the step.

Chapter 1 describes the format of job control statements. The operand field of the DD statement can contain keyword parameters that describe the location of the data set (for example, volume serial number and identification of the unit on which the volume will be mounted) and the attributes of the data itself (for example, record format).

The DD statement enables you to write PL/I source programs that are independent of the data sets and input/output devices they will use. You can modify the parameters of a data set or process different data sets without recompiling your program; for example, you can cause a program that originally read punched cards to accept input from magnetic tape merely by changing the DD statement.

Name of DD Statement

The name that appears in the name field of the DD statement (ddname) identifies the statement so that other job control statements and the PL/I program can refer to it. A ddname must be unique within a job step; if two DD statements in one job step have the same name, the second statement is ignored. Except when specifying the concatenation of two or more data sets, a DD statement must always have a name.

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

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

When read from a PL/I program the concatenated data sets need not be on the same volume, but the volumes must be on the same type of device, and the data sets must have similar characteristics (for example, block size and record format). You cannot process concatenated data sets backwards.

Parameters of DD Statement

The operand field of the DD statement contains keyword parameters that you can use to give the following information:

- The name of the data set (DSNAME parameter).
- Description of the device and volume that contain the data set (UNIT, VOLUME, SPACE, LABEL, and SYSOUT parameters).
- Disposition of the data set before and after execution of the job step (DISP parameter).
- Data set characteristics (DSC parameter).

The following paragraphs summarize the functions of these groups of parameters. For full details of all the parameters, refer to the job control language publications.

NAMING THE DATA SET

The DSNAME parameter specifies the name of a newly defined data set or refers to the name of an existing data set (for example, DSNAME=ROOTS). You need not specify the DSNAME parameter for a temporary data set (one that exists only for the duration of the job in which it is created); the operating system will give it a temporary name.

DEscribing the Device and Volume

The UNIT parameter specifies the type of input/output device to be allocated for the
The output device is any unit (but usually a device for example, \text{SPACE=(CYL,10)} specifying that 10 cylinders are to be allocated).

The \text{VOLUME} parameter identifies the volume on which the data set resides (for example, \text{VOLUME=SER=12345}). It can also include instructions for mounting and demounting volumes.

The \text{SPACE} parameter specifies the amount of auxiliary storage required to accommodate a data set on a direct-access device (for example, \text{SPACE=(CYL,10)} specifies that 10 cylinders are to be allocated).

The \text{LABEL} parameter specifies the type and contents of the data set labels for magnetic tape (for example, \text{LABEL=4} indicates that the data set is the fourth data set on the volume).

The \text{SYSOUT} parameter allows you to route an output data set through a system output device (for example, \text{SYSOUT=A}). A system output device is any unit (but usually a printer or a card punch) that is used in common by all jobs. The computer operator allocates all the system output devices to specific classes according to device type and function. The usual convention is for class A to refer to a printer and class B to a card punch; the IBM-supplied cataloged procedures assume that this convention is followed. If you use the \text{SYSOUT} parameter, the only other information you may have to supply about the data set is the block size, which you can specify either in the \text{DCB} parameter of the DD statement or in your PL/I program.

\section*{DISPOSITION OF THE DATA SET}

The \text{DISP} parameter indicates whether a data set exists or is new, and specifies what is to be done with it at the end of the job step (for example, \text{DISP=(NEW,KEEP)} specifies that a data set is to be created and that it is to be kept on the volume of the end of the job step). At the end of a job step, you can delete a data set, pass it to the next step in the same job, enter its name in the system catalog or have it removed from the catalog, or you can keep the data set for future use without cataloging it.

The \text{LEAVE} and \text{REREAD} options of the \text{ENVIRONMENT} attribute allow you to use the \text{DISP} parameter to control the action taken when the end of a magnetic-tape volume is reached or when a magnetic-tape data set is closed. For a description of these options refer to the language reference manual for this compiler.

\section*{Use of the Conditional Subparameters}

If you wish use the conditional subparameters of the \text{DISP} parameter for data sets processed by PL/I programs, the step abend facility must be used. The step abend facility is obtained as follows:

1. The \text{ERROR} condition should be raised or signaled whenever the program is to terminate execution after a failure that requires the application of the conditional subparameters.

2. The resident library subroutine \text{IBMABEND} must be changed to return a non-zero return code. The method of doing this is described in Chapter 12 under the heading "The ABEND Facility".

\section*{DATA SET CHARACTERISTICS}

The \text{DCB} (data control block) parameter of the DD statement allows you to describe the characteristics of the data in a data set, and the way it will be processed, at execution time. Whereas the other parameters of the DD statement deal chiefly with the identity, location, and disposal of the data set, the \text{DCB} parameter specifies information required for the processing of the records themselves. The subparameters of the \text{DCB} parameter are described in Appendix A. For \text{DCB} use, see "Data Control Block," later in this chapter.

The \text{DCB} parameter contains subparameters that describe:

- The organization of the data set and how it will be accessed (CYLOFL, DSORG, LIMCT, NCP, NTM, and OPTCD subparameters).
- Device dependent information such as the recording technique for magnetic tape or the line spacing for a printer (CODE, DEN, FUNC, MODE, PRTSP, STACK, and TRTCH subparameters).
- The record format (BLKSIZE, KEYLEN, LRECL, RECFM, and RKP subparameters).

Chapter 6: Data Sets and Files 79
• The number of buffers that are to be used (BUFNC subparameter).

• The printer or card punch control characters (if any) that will be inserted in the first byte of each record (RECFM subparameter).

You can specify BLKSIZE, BUFNO, LRECL, KEYLEN, NCP, RECFM, RKP, and TRIOFL (or their equivalents) in the ENVIRONMENT attribute of a file declaration in your FL/I program instead of in the DCB parameter.

You cannot use the DCB parameter to override information already established for the data set in your FL/I program (by the file attributes declared and the other attributes that are implied by them). DCB subparameters that attempt to change information already supplied are ignored. An example of the DCB parameter is:

```
DCB=(RECFM=FB,BLKSIZE=400,LRECL=40)
```

This parameter specifies that fixed-length records, 40 bytes in length, are to be grouped together in a block 400 bytes long.

### Operating System Data Management

An object module produced by the optimizing compiler uses the data management routines of the operating system to control the storage and retrieval of data. The compiler translates each input and output statement in a PL/I program into a set of machine instructions that result in the issuing of assembler language macro instructions that request the data management routines to perform the required input or output operations. (These macro instructions are described in the supervisor and data management macro instructions publication.)

The macro instructions are issued either directly, by compiled code, or by appropriate subroutines from the transient library. In the latter case, the compiled code includes a branch to an interface subroutine in the resident library that initiates the flow of control through other required library subroutines.

The data management routines control the organization of data sets, as well as the storage and retrieval of the records they contain. They create and maintain data set labels, indexes, and catalogs; they transmit data between main storage and auxiliary storage; they use the system catalog to locate data sets; and they request the operator to mount and demount volumes as required.

### Buffers

The data management routines can provide areas of main storage, termed buffers, in which data can be collected before it is transmitted to auxiliary storage, or into which it can be read before it is made available to a program. The use of buffers permits the blocking and de-blocking of records, and may allow the data management routines to increase the efficiency of transmission of data by anticipating the needs of a program. Anticipatory buffering requires at least two buffers: while the program is processing the data in one buffer, the next block of data can be read into another. Anticipatory buffering can only be used for data sets being accessed sequentially.

The operating system can further increase the efficiency of transmission in a program that involves many input/output operations by using chained scheduling. In chained scheduling, a series of read or write operations are chained together and treated as a single operation. For chained scheduling to be effective, at least three buffers are necessary. For more information on chained scheduling see the data management services publication.

The data management routines have two ways of making data that has been read into a buffer available to a program. In the move mode, the data is actually transferred from the buffer into the area of main storage occupied by the program. In the locate mode, the program can process the data while it is still in the buffer; the data management routines pass the address of the buffer to the program to enable it to locate the data. Similarly, a program can move output data into the buffer or it can build the data in the buffer itself.

### Access Methods

Data management has two access techniques for transmitting data between main storage and auxiliary storage:

The queued access technique deals with individual records, which it blocks and deblocks automatically. Records are retrieved and written by means of macro instructions. The first time a macro instruction is issued to retrieve a record, the data management routines place a block...
of records in an input buffer and pass the first record to the program that issued the instruction (that is, they deblock the records); each succeeding retrieval passes another record to the program. When the input buffer is empty, it is automatically refilled with another block. Similarly, another macro instruction places records in an output buffer and, when the buffer is full, writes out the records. Since the queued access technique brings records into main storage before they are requested, it can be used only for records that have been stored sequentially.

The basic access technique uses other macro instructions for input and output. These instructions move blocks, not records. When a macro instruction is issued to retrieve a block, the data management routines pass a block of data to the program that issued the instruction; they do not deblock the records. Similarly, another macro instruction transmits a block to auxiliary storage.

The combination of data set organization, as described earlier in this chapter, and an access technique is termed an access method. The access methods used by the compiler are shown in Figure 6-5.

**Figure 6-5. The access methods used by the compiler**

- **QSAM:** Queued sequential access method: this combines the queued access technique with sequential organization.
- **QISAM:** Queued indexed sequential access method: this combines the queued access technique with indexed sequential organization.
- **BSAM:** Basic sequential access method: this combines the basic access technique with sequential organization.
- **BISAM:** Basic indexed sequential access method: this combines the basic access technique with indexed sequential organization.
- **BDAM:** Basic direct access method: this combines the basic access technique with direct organization.
- **TCAM:** Telecommunications access method: this combines the queued access technique with telecommunications organization.

The PL/I library subroutines use QSAM for stream-oriented transmission and all of the above methods for record-oriented transmission, as shown in Figure 6-6. They implement PL/I GET and PUT statements by transferring the appropriate number of characters from or to the buffers, and use GET and PUT macro instructions in the locate mode to fill or empty the buffers. (For paper tape, however, the library subroutines use the move mode to permit translation of the transmitted characters before passing them to the PL/I program.)

**DATA CONTROL BLOCK**

A data control block (DCB) is an area of main storage that contains information about a data set and the volume that contains it. The data management routines refer to this information when they are processing a data set; no data set can be processed unless there exists a corresponding DCB. For a PL/I program, a PL/I library subroutine creates a DCB for the associated data set when a file is opened.

A data control block contains two types of information: data set characteristics and processing requirements. The characteristics include record format, record length, block size, and data set organization. The processing information may specify the number of buffers to be used, and it may include device-dependent information (for example, printer line spacing or magnetic-tape recording density), and special processing options that are available for some data-set organizations.

The information in the DCB comes from three sources:

1. The file attributes declared implicitly or explicitly in the PL/I program.
2. The data definition (DD) statement for the data set.
3. If the data set exists, the data set labels.

**OPENING A FILE**

The execution of a PL/I OPEN statement associates a file with a data set. This requires the merging of the information describing the file and the data set. If any conflict is detected between file...
### Figure 6-6. Access methods for record-oriented transmission

<table>
<thead>
<tr>
<th>Data Set Organization</th>
<th>File Attributes</th>
<th>Access Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONSECUTIVE</td>
<td>SEQUENTIAL</td>
<td>BUFFERED</td>
</tr>
<tr>
<td></td>
<td>INPUT</td>
<td>QSAM</td>
</tr>
<tr>
<td></td>
<td>OUTPUT</td>
<td>BSAM</td>
</tr>
<tr>
<td></td>
<td>UPDATE</td>
<td></td>
</tr>
<tr>
<td>INDEXED</td>
<td>SEQUENTIAL</td>
<td>BUFFERED</td>
</tr>
<tr>
<td></td>
<td>INPUT</td>
<td>QISAM</td>
</tr>
<tr>
<td></td>
<td>OUTPUT</td>
<td>BISAM</td>
</tr>
<tr>
<td></td>
<td>UPDATE</td>
<td></td>
</tr>
<tr>
<td>REGIONAL</td>
<td>SEQUENTIAL</td>
<td>BUFFERED</td>
</tr>
<tr>
<td></td>
<td>INPUT</td>
<td>BSAM</td>
</tr>
<tr>
<td></td>
<td>OUTPUT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>UPDATE</td>
<td>BISAM</td>
</tr>
<tr>
<td>TELEPROCESSING</td>
<td>TRANSIENT</td>
<td>BUFFERED</td>
</tr>
<tr>
<td></td>
<td>INPUT</td>
<td>TCAM</td>
</tr>
<tr>
<td></td>
<td>OUTPUT</td>
<td></td>
</tr>
</tbody>
</table>

#### PL/I PROGRAM

```pli
DCL MASTER FILE ENV(FB BLKSIZE(400), RECSIZE(40));
OPEN FILE(MASTER);
```

#### DD STATEMENT

```plaintext
//MASTER DD UNIT=2400
VOLUME=SER=1791,
DSNAME=LIST,
DCB=(BUFNO=3,
RECFM=F,
BLKSIZE=400,
LRECL=100)
```

#### DATA SET LABEL

- Record format=F
- Record length=100
- Blocking factor=4
- Recording density=1600

#### DATA CONTROL BLOCK

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Record format</td>
<td>FB</td>
</tr>
<tr>
<td>Block size</td>
<td>400</td>
</tr>
<tr>
<td>Record length</td>
<td>40</td>
</tr>
<tr>
<td>Device type</td>
<td>2400</td>
</tr>
<tr>
<td>Number of buffers</td>
<td>3</td>
</tr>
<tr>
<td>Recording density</td>
<td>1600</td>
</tr>
</tbody>
</table>

Note: Information from the PL/I program overrides that from the DD statement and the data set label. Information from the DD statement overrides that from the data set label.

#### Figure 6-7. How the operating system completes the DCB
attributes and data set characteristics the UNDEFINEDFILE condition will be raised.

Subroutines of the PL/I library create a skeleton data control block for the data set, and use the file attributes from the DECLARE and OPEN statements, and any attributes implied by the declared attributes, to complete the data control block as far as possible, as shown in Figure 6-7. They then issue an OPEN macro instruction, which calls the data management routines to check that the correct volume is mounted and to complete the data control block. The data management routines examine the data control block to see what information is still needed and then look for this information, first in the DD statement, and finally, if the data set exists and has standard labels, in the data set labels. For new data sets, the data management routines begin to create the labels (if they are required) and to fill them with information from the data control block.

Neither the DD statement nor the data set label can override information provided by the PL/I program; nor can the data set label override information provided by the DD statement.

When the DCB fields have been filled in from these sources, control returns to the PL/I library subroutines. If any fields have still not been filled in, the PL/I OPEN subroutine provides default information for some of them; for example, if LRECL has not been specified, it is now provided from the value given for BLKSIZE.

CLOSING A FILE

The execution of a PL/I CLOSE statement dissociates a file from the data set with which it was associated. The PL/I library subroutines first issue a CLOSE macro instruction and, when control returns from the data management routines, release the data control block that was created when the file was opened. The data management routines complete the writing of labels for new data sets and update the labels of existing data sets.

Auxiliary Storage Devices

The following paragraphs summarize the salient operational features of various types of auxiliary storage devices.

IBM 2520 AND 2540 CARD READER AND PUNCH

Both the card reader and card punch accept F-format, V-format, and U-format records; the control bytes of V-format records are not punched. Any attempt to block records is ignored.

Each punched card corresponds to one record; you should therefore restrict the maximum record length to 80 bytes (EBCDIC mode) or 160 bytes (column-binary mode). To select the mode, use the MODE subparameter of the DCB parameter of the DD statement; if you omit this subparameter, EBCDIC is assumed. (The column-binary mode increases the packing density of information on a card, punching two bytes in each column. Only six bits of each byte are punched; on input, the two high-order bits of each byte are set to zero; on output, the two high-order bits are lost.) The Card Read Punch 2540 has five stackers into which cards are fed after reading or punching. Two stackers accept only cards that have been read, and two others accept only those that have been punched; the fifth (center) stacker can accept either cards that have been read or those that have been punched. The two stackers in each pair are numbered 1 and 2 and the center stacker is numbered 3, as shown in Figure 6-8.

Figure 6-8. Card read punch 2540: stacker numbers

The Card Read Punch 2520 has two stackers, into which cards can be read or punched. The Card Reader 2501 has only one stacker.

Cards are normally fed into the appropriate stacker 1 after reading or punching. You can use the STACR subparameter of the DCB parameter of the DD statement to select an alternative stacker for reading or punching. For punching only, you can select the stacker dynamically by inserting an ANS or System/360 code in the first byte of each record; you must indicate which code you are using in the RECFM subparameter of the DD statement or in the ENVIRONMENT option.
The control character is not punched.

IBM 3505 AND 3525 CARD READER AND PUNCH

The 3505 Card Reader and the 3525 Card Punch are available only to System/370 users. These two devices are functionally separate and operate independently of each other.

The 3505 will read 80-column cards, and provides, in addition to normal card reading, the following facilities:

- Optical mark read (in EBCDIC or column binary mode).
- Read column eliminate (in EBCDIC or column binary mode).
- Stacker selection.

The 3525 is basically an 80-column card punch, and can have the following additional facilities:

- Card reading facilities that optionally include:
  
  Reading in EBCDIC or column binary mode.
  
  Read column eliminate.

- Card punching in EBCDIC or column binary mode.

- Card printing facilities that include either:
  
  Two-line printing.
  
  or:
  
  Multiline printing (up to 25 lines).

- Punch interpretation.

- Stacker selection.

The various operations of the 3505 and the 3525 are described in the following sections. In general, the operations to be performed are selected by the FUNC, MODE, and STACK subparameters of the DCB parameter. The formats of these subparameters are described in Appendix A of this manual.

Basic Card Reading and Punching

Card reading or punching on a 3525 is selected by specifying DCB=(FUNC=R) for reading or DCB=(FUNC=P) for punching. If the FUNC subparameter is not specified, the default is FUNC=R for input files and FUNC=P for output files that do not have the PRINT attribute.

Apart from this function selection for the 3525, support for the 3505 as a simple card reader and the 3525 as a card reader or punch is identical to that for the 2540 described earlier in this chapter.

EBCDIC or Column Binary Modes

Cards processed by a 3505 or a 3525 can hold data coded in either EBCDIC or column binary mode. If EBCDIC is used, each card can contain up to 80 characters. If column binary 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 low-order bit appears in row 12 of the card column for the first character, and in row 4 for the second character. The binary values of characters are transmitted to successive bytes in main storage. The two high-order bits of each byte are set to zero (these bits are not represented in the 6-bit code). The characters are transmitted in the order: first (top) character, second (bottom) character, and so on for each column in the card, from column 1 to column 80.

The details of the coding and conversion technique used for column binary data are left to the program designer. The TRANSLATE built-in function may provide a convenient method of converting data to or from column binary form.

Rules for using column binary mode are:

- The MODE subparameter of the DCB parameter must specify column binary (MODE=C).
- The PL/I file must have the RECORD attribute.
- The punch-interpret feature must not be used.
- The file must be either an input file or an output punch file. It cannot be a print file.
A column binary output file must have a record size of 160 bytes.

Stacker Selection

The stacker selection feature is optionally available on the 3505 and is a standard feature on a 3525. There are two methods of selecting a stacker:

1. The stacker can be selected permanently for all cards in the file. This method involves the STACK subparameter of the DCB parameter.

2. 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.

Optical Mark Read

The optical mark read (OMR) feature is available only on the 3505 card reader. This feature enables preprinted or pencil-written marks on a punched card to be read as data. The following rules apply:

- Optical mark read is specified by MODE=EO (EBCDIC mode) or MODE=CO (column binary mode) in the DCB parameter.

- The associated PL/I file must have the RECORD and INPUT attributes, and must not have the TOTAL attribute.

- Records must be F-format with a RECSIZE of 80 (EBCDIC mode) or 160 (column binary mode).

- Up to 40 columns of EBCDIC data or 80 characters of column binary data can be read optically from a single card. Optical and punched data can be read from the same card although there are some restrictions, given below, on how the data is recorded on the card.

- Optical mark data can appear only in alternate card columns and must be separated by blank columns. Optical mark and punched hole columns must also be separated by at least one blank column. When the record is read in, the data is compressed by removing the blank column following each optical mark column, and the record is padded with blanks.

- The columns containing optically-readable marks must be specified to the program at execution-time by a format descriptor card. This card must be the first card in the deck of cards to be read by the file each time the program is run. Operating procedures for running jobs that use OMR should ensure that this point is not overlooked.

- The OMR descriptor card has the following format:

  FORMAT (n1,n2),(n3,n4)...

  where n1 is the first column in a group to be read in OMR mode, n2 is the last column in the group, n3 is the first column in the next group, n4 is the last column in this group, and so on. Remember that only every other column between n1 and n2 or n3 and n4 can be read in OMR mode. A maximum of 40 columns of OMR data can be accommodated on an 80-column card. n1 and n2 (and similarly n3 and n4) must be either both even or both odd, and n3 must be at least 2 greater than n2.

  The format descriptor record must begin in column 2 and can continue through column 71. If a continuation is required, punch any character in column 72 and start the continuation in column 16 of the following card.

  A blank must follow the keyword FORMAT. Operands must be separated by a comma. 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.

  Note that column 1 of the card always corresponds with the first byte of the data in main storage. Consequently, if an optical mark appears in column 2, column 1 must be blank and byte 1 will also be blank.

  If a marginal mark, weak mark, or poor erasure is detected on a column, the corresponding byte and the last byte of the record are set to X'3F'. The TRANSMIT condition is raised once only for all errors found in a card. The card itself is stacked in the alternative stacker to that normally used by the file.

  When an optical mark read file is closed, the last card is fed and stacked in the same stacker as the previous
card. This feed operation resets the device into unformatted mode, ready for the next file opening.

• Optical mark read is not supported on SYSIN. The 3505 must be allocated exclusively to the user's job by specifying the device type of the unit address in the UNIT parameter of the DD statement.

• When a file is opened for optical mark reading, the value of the BUFFERS option (for BUFFERED files) or the NCP option (for UNBUFFERED files) is automatically set to 1.

Read Column Eliminate

The read column eliminate (RCE) feature is optionally available on the 3505 and on a 3525 with card reading facilities. This feature permits the selective reading of card columns. The columns to be ignored when the card is read are specified in a format descriptor card. The ignored columns are replaced by blanks in EBCDIC mode or zeroes in column binary mode before the record is transmitted.

The following rules apply:

• Read column eliminate is specified by MODE=ER (EBCDIC mode) or MODE=CR (column binary mode) in the DCB parameter.

• An RCE format descriptor card must be supplied. This card must be the first card in the deck of cards to be read by the program each time it is executed. Operating procedures for running jobs that use RCE should ensure that this point is not overlooked.

• The RCE descriptor card has the following format:

  FORMAT (n1,n2),(n3,n4)...

  where n1 is the first column in a group of columns to be ignored and n2 is the last column in the group, n3 is the first column in the next group to be ignored, n4 is the last column in this group, and so on.

  The format descriptor card must begin in column 2 and continue through to column 71. If a continuation is required, punch any character in column 72 and start the continuation in column 16 of the following card.

  A blank must follow the keyword FORMAT. Operands must be separated by a comma.

  Example:

  FORMAT (20,30),(52,76)

  This specifies that columns 20 through 30 and columns 52 through 76 are to be ignored when the card is read.

• The PL/I file can have either the STREAM or the RECORD attribute. Records must be F-format.

• When an RCE file is closed, a card feed operation is executed by the reader. If several files are to be read consecutively - either for successive programs in a single batch, or for several files in a single program - a non-data card must separate the files.

• Read column eliminate is not supported on SYSIN. The 3505 or 3525 must be allocated exclusively to the user's job by specifying the device type of the unit address in the UNIT parameter of the DD statement.

Punch Interpret

A single file can be used to punch and interpret cards by specifying DCB=(FUNC=I). Cards are punched normally, and the same data is printed on lines 1 and 3 of the card. The first 64 characters are printed on line 1; the remaining 16 characters are right-justified on line 3.

A punch interpret file may have the STREAM or RECORD and the BUFFERED or UNBUFFERED attributes. Records must be F format, with a record size of 80, or 81 if control characters are being used for stacker selection.

Printing on Cards

The card printing feature of the 3525 is available in two forms:

1. Two-line printing.

2. Multiline printing (up to 25 lines).

Printing can be performed either as the only operation on the card, or as one of a number of operations on the same card. The following rules apply to print-only files. The additional requirements for printing after reading or punching a card are described under "Multiple Operations" later in this chapter.
• The FUNC subparameter of the DCB parameter must specify "W" if the 3525 has the multiline print feature, or "WT" if it has the 2-line print feature. If FUNC is omitted, FUNC=W is defaulted for PL/I PRINT files.

• The PL/I file may have either the RECORD or the STREAM attribute.

• The maximum number of characters that can be printed on each line is 64. The user must ensure that this limit is not exceeded; in particular, on PRINT files, LINESIZE should not exceed 64 or data will be lost.

• If the 3525 has the two-line print feature, and is used by a file with the PRINT attribute or by a file using CTLASA or CTL360 control characters, care should be taken to ensure that no attempt is made to print on any line other than lines 1 and 3. Such an attempt will cause the program to be terminated without raising the PL/I ERROR condition. If a PRINT file is used, and a PAGESIZE of more than 3 is specified, the pagesize is set to 3 when the file is opened.

If the file is a non-PRINT file, and control characters are not used, records are printed on lines 1 and 3 automatically.

• If a 3525 with the multiple print feature is used, the file should have a maximum pagesize of 25. If a PAGESIZE of greater than 25 is specified on a PRINT file, the pagesize is set to 25 when the file is opened. Whatever the page size, a PUT PAGE statement for a PRINT file will always cause the file to be positioned at line 1 of the next card. Any attempt to print beyond line 25 will cause the program to be terminated without raising the PL/I ERROR condition.

• All the standard ASA control characters can be used, with the exception of "+" (suppress space before printing). The use of the "+" control character, or of SKIP(0) on a PRINT file, will cause the program to be terminated without raising the PL/I ERROR condition.

Odd numbered lines on a card can be reached using "skip to channel" control characters, channel numbers being defined as follows:

channel number = (line number + 1)/2

Only channels 1 through 12 are valid. Other lines can be reached by using "space and print" control characters.

All lines can be reached by executing sufficient WRITE or PUT operations.

Multiple Operations

Two or three files may be used in association with each other to enable more than one of the operations "read", "punch", and "print" to be performed on a single card during one pass through a 3525. A DD statement is required for each operation that the device is to perform, and the association of these data sets is specified by means of the unit affinity (AFF) parameter, together with the FUNC subparameter of the DCB parameter.

For example, for as set of files that are to perform the operations read-punch-print the association of the data sets and the set of operations are specified as follows:

//CARDIN DD UNIT=3525,DCE=(FUNC=RPW)
//PUNCH DD UNIT=AFF=CARDIN,DCE=(FUNC=RPW)
//PRINT DD UNIT=AFF=PUNCH,DCE=(FUNC=RPWX)

Valid FUNC options are listed in Appendix A of this manual. Note that the FUNC option must specify the complete set of associated operations. "X" must be added to the FUNC option of the print data set. If the 3525 has the two-line print feature, T must also be coded on the FUNC option of the print data set.

The following rules apply to multiple operations:

• All the device-associated files must have the RECORD attribute, and must be all BUFFERED or all UNBUFFERED. None of the files can have the TOTAL option. Records must be F-format.

• If stacker selection is required, it can only be specified on the punch file, if there is one. Either stacker-select control characters or static stacker selection by means of the STACK subparameter can be used.

• An associated data set cannot be allocated to SYSIN or SYSCUT. The 3525 must be allocated exclusively to the user's job by specifying the device type of the unit address in the UNIT parameter of the DD statement.

• Data delimiter cards should not be punched or printed on, or the first card of the following job will be lost.

Details of how to open and close associated files, and of the sequences of
operations that can be performed, are given in the Language Reference Manual for this compiler.

Data Protection

To avoid erroneous punching into card columns that already contain data, a "data protection" option can be used on a punch file which is in association with a read file. Data Protection is specified by a "D" in the FUNC option of the DD statement for the punch data set. The user must provide an 80-byte data protection image (DPI) and linkedit it into SYS1.IMAGELIB with a member name of the form FORMxxxx. The DPI contains blanks in columns that are to be protected, and any alphanumeric character in columns that can be punched. An assembler language program is used to link edit the DPI; an example is given in Figure 6-9.

```assembler
//UP EXEC ASMFCL
//ASM.SYSIN DD *
FORMDPI CSECT
DC X'40' (protected column)
DC X'40' (protected column)
DC C'1356789A' (punch columns)
DC 70X'40' (protected columns)
END
/*
//LKED.SYSLMOD DD DISP=OLD,
// DNAME=SYS1.IMAGELIB(FORMxxxx)
Figure 6-9. An example of a program to link edit the DPI

A particular DPI is selected by means of the FCB parameter of the DD statement for the punch file. For example:

//PUNCH DD UNIT=AFF=CARDIN,
// DCB=(FUNC=RPWD),
// FCB=xxxx

Data protection cannot be specified for column binary cards.

PRINTER

The printer accepts F-format, V-format, and U-format records; the control bytes of V-format records are not printed. Each line of print corresponds to one record; you should therefore restrict your record length to the length of one printed line. Any attempt to block records is ignored.

You can use the PRTSP subparameter of the DCB parameter of the DD statement to request the line spacing of your output, or you can control the spacing dynamically by inserting an ANSI or System/360 code in the first byte of each record; you must indicate which code you are using in the RECFM subparameter of the DD statement or in the ENVIRONMENT option. The control character is not printed. If you do not specify the line spacing, single spacing (no blanks between lines) is assumed.

MAGNETIC TAPE

Magnetic-tape devices accept D-format, F-format, and U-format records for both 9-track and 7-track magnetic tape with the one exception that 7-track magnetic tape will not accept V-format records unless the data conversion feature is available. (The data in the control bytes of V-format records is in binary form; in the absence of the data conversion feature, only six of the eight bits in each byte are transmitted to 7-track tape.)

Nine-track magnetic tape is standard in IBM System/360, but some 2400 series magnetic-tape drives incorporate features that facilitate reading and writing 7-track tape. The translation feature changes character data from EBCDIC (the 8-bit code used in System/360) to BCD (the 6-bit code used on 7-track tape) or vice-versa. The data conversion feature treats all data as if it were in the form of a bit string, breaking the string into groups of eight bits for reading into main storage, or into groups of six bits for writing on 7-track tape; the use of this feature precludes reading the tape backwards. To use translation or data conversion, include the TRCH (tape recording technique) subparameter in the DCB parameter of the DD statement.

The maximum recording density available depends on the model number of the tape drive; single-density tape drive units have
a maximum recording density of 800 bytes per inch, and dual-density tape drive units have a maximum of 1600 bytes per inch. For 9-track tapes, a single-density drive offers only the 800 bytes per inch density; the standard density for a dual-density drive is 1600 bytes per inch, but you can use the subparameter DEN (density) of the DD statement to specify 800 bytes per inch. For 7-track tape, the standard recording density for both types of drive unit is 200 bytes per inch; you can use the DEN subparameter of the DCE parameter of the DD statement to select alternatives of 556 or 800 bytes per inch.

Note: When a data check occurs on a magnetic-tape device with short length records (12 bytes on a read and 18 bytes on a write), these records will be treated as noise.

DIRECT-ACCESS DEVICES

Direct-access devices accept F-format, V-format, and U-format 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 2311 disk pack has ten recording surfaces, each of which has 200 concentric tracks; thus, it contains 200 cylinders, each of which contains ten tracks.

When you create a data set on a direct-access device, you must always indicate to the operating system how much auxiliary storage the data set will require. Use the SPACE parameter of the DD statement to allocate space in terms of blocks, tracks, or cylinders. If you request space in terms of tracks or cylinders, bear in mind that space in a data set on a direct-access device is occupied not only by blocks of data, but by control information inserted by the operating system; if you use small blocks, the control information can result in a considerable space overhead. The following reference cards contain tables that will enable you to determine the amount of space you will require:

- 2301 Drum Storage Unit, Order No. GX20-1717
- 2302 Disk storage Drive, Order No. GX20-1706
- 2303 Drum Storage Unit, Order No. GX20-1718
- 2311 Disk Storage Drive, Order No. GX20-1705
- 2314 Storage Facility, Order No. GX20-1710
- 2321 Data Cell Drive, Order No. GX20-1704
- 3330 Series Disk Storage, Order No. GX20-1920
This chapter describes how to define data sets for use with PL/I files that have the STREAM attribute. It explains how to create and access data sets with CONSECUTIVE organization. The essential parameters of the DD statements used in creating and accessing these data sets are summarized in tables, and several examples of PL/I programs (complete with JCL) are included to illustrate the text.

Data sets with the STREAM attribute are processed by stream-oriented transmission, which allows the PL/I program to ignore block and record boundaries and treat a data set as a continuous stream of data items in character form.

For output, the data management subroutines of the PL/I library convert the data items from the program variables into character form if necessary, and build the stream of characters into records for transmission to the data set.

For input, the library subroutines take records from the data set and separate them into the data items requested by the program, converting them into the appropriate form for assignment to the program variables.

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.

Creating a Data Set

Any data set created using stream-oriented transmission must have CONSECUTIVE organization, but it is not necessary to specify this in the ENVIRONMENT attribute, since it is the default organization.

Your program deals only with data items, and not with records and blocks as they will exist in the data set. Accordingly, you need not concern yourself with the actual structure of the data set beyond specifying a block size (which is always necessary), unless you propose to use record-oriented transmission to access the data set at a later date.

To create a data set, you must give the operating system certain information either in your PL/I program or in the DD statement that defines the data set. The following paragraphs indicate the essential information, and discuss some of the optional information you may supply; the ENVIRONMENT attribute and the LINESIZE option are discussed fully in the language reference manual for this compiler.

ESSENTIAL INFORMATION

You must supply the following information, summarized in Figure 7-1, when creating a data set:

- Device that will write or punch your data set (UNIT, SYSOUT, or VOLUME parameter of DD statement).
- Block size: you can specify the block size either in your PL/I program (ENVIRONMENT attribute or LINESIZE option) or in the DD statement (ELKSIZE subparameter). If you do not specify a record length, unblocked records are assumed and the record length is determined from the block size. If you do not specify a record format, U-format is assumed (except for PRINT files when V-format is assumed: see "PRINT Files," later in this chapter).

If you want to keep a magnetic-tape or direct-access data set (that is, you do not want the operating system to delete it at the end of your job), the DD statement must name the data set and indicate how it is to be disposed of (DSNAME and DISP parameters). The DISP parameter alone will suffice if you want to use the data set in a later step but will not need it after the end of your job.

When creating a data set on a direct-access device, you must specify the amount of space required for it (SPACE parameter of DD statement).

If you want your data set stored on a particular magnetic-tape or direct-access device, you must indicate the volume serial number in the DD statement (SER or REF subparameter of VOLUME parameter). If you do not supply a serial number for a magnetic-tape data set that you want to keep, the operating system will allocate one, inform the operator, and print the number on your program listing.

If your data set is to follow another
data set on a magnetic-tape volume, you must use the LABEL parameter of the DD statement to indicate its sequence number on the tape.

**Example**

The use of stream-oriented transmission to create a data set on a 2311 disk drive is shown in Figure 7-2. The data read from the input stream by the standard file SYSIN includes a field VREC that contains five unnamed 7-character subfields; the field NUM defines the number of these subfields that contain information. The output file WORK transmits to the data set the whole of the field VREC and only those subfields of VREC that contain information. The data set has U-format unblocked records with a maximum block size of 400 bytes (defined in the statement that declares the file WORK). All blocks except the last will contain exactly 400 bytes.

**Accessing a Data Set**

A data set accessed using stream-oriented transmission need not have been created by stream-oriented transmission, but it must have CONSECUTIVE organization, and all the data in it must be in character form. You can open the associated file for input, and read the records the data set contains; or you can open the file for output, and extend the data set by adding records at the end.

To access a data set, you must identify it to the operating system in a DD statement. The following paragraphs, which are summarized in Figure 7-3, indicate the essential information you must include in the DD statement, and discuss some of the optional information you may supply. The discussions do not apply to data sets in the input stream, which are dealt with in Chapter 6.
**ESSENTIAL INFORMATION**

If the data set is cataloged, you need only supply the following information in the DD statement:

- The name of the data set (DSNAME parameter). The operating system will locate the information describing the data set in the system catalog, and, if necessary, will request the operator to mount the volume containing it.

- Confirmation that the data set exists (DISP parameter). If you open the data set for output with the intention of extending it by adding records at the end, code DISP=MOD; otherwise, to open the data set for output will result in its being overwritten.

If the data set is not cataloged, you must, in addition, specify the device that will read the data set and, for magnetic-tape and direct-access devices, give the serial number of the volume that contains the data set (UNIT and VOLUME parameters).

If the data set is on paper tape or punched cards, you must specify the block size either in your PL/I program (ENVIRONMENT attribute) or in the DD statement (BLKSIZE subparameter).

If the data set follows another data set on a magnetic-tape volume, you must use the LABEL parameter of the DD statement to indicate its sequence number on the tape.

**MAGNETIC TAPE WITHOUT STANDARD LABELS**

If a magnetic-tape data set has nonstandard labels or is unlabeled, you must specify the block size either in your PL/I program (ENVIRONMENT attribute) or in the DD statement (BLKSIZE subparameter). The DSNAME parameter is not essential if the data set is not cataloged.

PL/I data management includes no facilities for processing nonstandard labels, which, to the operating system, appear as data sets preceding or following your data set. You can either process the
<table>
<thead>
<tr>
<th>Parameters of DD Statement</th>
<th>When required</th>
<th>What you must state</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>All devices</td>
<td>Name of data set</td>
<td>DSNAME=</td>
<td></td>
</tr>
<tr>
<td>Input device</td>
<td>Disposition of data set</td>
<td>DISP=</td>
<td></td>
</tr>
<tr>
<td>Standard labeled</td>
<td>All devices</td>
<td>UNIT= or VOLUME=REF=</td>
<td></td>
</tr>
<tr>
<td>Magnetic tape and direct access</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sequence number</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LABEL=</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Block size</td>
<td></td>
<td>DCB=(BLKSIZE=...)</td>
<td></td>
</tr>
</tbody>
</table>

*Alternatively, you can specify the block size in your PL/I program by using either the ENVIRONMENT attribute or the LINESIZE option.

Figure 7-3. Accessing a data set: essential parameters of DD statement

```pli
//OPT7#4 JOB
//STEP1 EXEC PLIXCLG
//PLI.SYSIN DD *
PEOPLE: PROC OPTIONS(MAIN);

DCL WORK FILE STREAM INPUT,
1 REC,
  2 FREC,
    3 NAME CHAR(19),
    3 NUM CHAR(1),
    3 SERNO CHAR(7),
    3 PROF CHAR(18),
    2 VREC CHAR(35),
  IN CHAR(80) DEF REC;

ON ENDFILE(WORK) GO TO FINISH;

OPEN FILE(WORK);
MORE:
  GET FILE(WORK) EDIT(IN,VREC) (A(45),A(7*NUM));
  PUT FILE(SYSPRINT) SKIP EDIT(IN) (A);
  GO TO MORE;
FINISH: CLOSE FILE(WORK);
END PEOPLE;
/*
//GO.WORK DD DSN=PEOPLE,UNIT=2311,VOL=SER=D186,DISP=(OLD,KEEP)

Figure 7-4. Accessing a data set with stream-oriented transmission
```
labels as independent data sets or use the
LABEL parameter of the DD statement to
bypass them; to bypass the labels code
LABEL=(2,NL) or LABEL=(,BLP).

RECORD FORMAT

When using stream-oriented 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 to your program from the data
stream.

If you do give record format
information, it must be compatible with the
actual structure of the data set. For
example, if a data set is created with F­
format records, a record size of 600 bytes,
and a block size of 3600 bytes, you can
access the records as if they are U-format
with a maximum block size of 3600 bytes;
but if you specify a block size of 3500
bytes, your data will be truncated.

EXAMPLE

The program in Figure 7-4 reads the data
set created by the program in Figure 7-2
and uses the standard 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, which always
contains 35 characters. At each execution
of the GET statement, VREC consists of the
number of characters generated by the
expression 7*NUM, together with sufficient
blanks to bring the total number of
characters to 35. The DISP parameter of the
DD statement could read simply DISP=OLD; if
the second term is omitted, an existing
data set will not be deleted.

Print Files

Both the operating system and the PL/I
language include features that facilitate the
formatting of printed output. The
operating system allows you to use the
first byte of each record for a printer
control code; the control codes, which are
not printed, cause the printer to skip to a
new line or page. Tables of printer control
codes are given in Figures 8-5 and 8-6. In
a PL/I program, the use of a PRINT file
provides a convenient means of controlling
the layout of printed output in stream­
oriented transmission; the compiler
automatically inserts printer control codes
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 printer control codes 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 ANS printer control code, and inserts
the appropriate codes automatically. A
PRINT file uses only the five printer
control codes shown in Figure 1-5.

<table>
<thead>
<tr>
<th>Code</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>b (blank)</td>
<td>Space 1 line before printing</td>
</tr>
<tr>
<td>0</td>
<td>Space 2 lines before printing</td>
</tr>
<tr>
<td>-</td>
<td>Space 3 lines before printing</td>
</tr>
<tr>
<td>+</td>
<td>No space before printing</td>
</tr>
<tr>
<td>1</td>
<td>Start new page</td>
</tr>
</tbody>
</table>

Figure 7-5. Printer control codes used
by a PRINT file

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 code
in the next record. If SKIP or LINE
specifies more than a three line space, the
compiler inserts sufficient blank records
with appropriate control codes to
accomplish the required spacing. In the
absence of a printer control option or
format item, when a record is full the
compiler inserts a blank code (single line
space) in the first byte of the next
record.

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), in a DD
statement, or by giving a line size in an
OPEN statement (LINESIZE option). The
record length must include the extra byte
for the printer control code, that is, it
must be one byte larger than the length of
the printed line (five bytes larger for V­
format records). The value you specify in
the LINESIZE option refers to the number of
SINE: PROC OPTIONS(MAIN);

DCL TABLE FILE STREAM OUTPUT PRINT,
TITLE CHAR(13) INIT('NATURAL SINES'),
HEADINGS CHAR(90) INIT( 0  6 12 18
24 30 36 42 48 54'),
PGNO FIXED DEC(2) INIT(1),
FINISH BIT(1) INIT('0'B),
VALUES(0:359,0:9) FLOAT DEC(6);

ON ENDPAGECTABLE) BEGIN;
PUT FILE(TABLE) EDITC'PAGE',PGNO)LINE(55),COL(87),A,F(3));
IF FINISH='0'B THEN DO;
PGNO=PGNO+1;
PUT FILE(TABLE) EDIT(TITLE||'( CONT'SD'),HEADINGS) 
(PAGE,A,SKIP(3),A);
END;
END;

DO I=0 to 359;
DO J=0 TO 9;
VALUES(I,J)=SIND(I+J/10);
END;
END;

OPEN FILE(TABLE) PAGESIZE(52) LINESIZE(93);
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);
CLOSE FILE(TABLE);
END SINE;

/*
GO.TABLE DD DSN=SINES,UNIT=2311,DISP=(NEW,CATLG),VOI=SER=D186,
SPACE=(CYL,(1,1))
*/

Figure 7-6. Creating a data set using a PRINT file

characters in the printed line; the compiler adds the printer control bytes.

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 nine 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 nine bytes greater than the maximum line size you intend to use, or ensure that the first OPEN statement specifies the maximum line size. (Output destined for the printer may be stored temporarily on a direct-access device,
unless you specify a printer by using
UNIT=, even if you intend it to be fed
directly to the printer.)

Since PRINT files have a default line
size of 120 characters, you need not give
any record format information for them. In
the absence of other information, the
compiler assumes V-format records; the
complete default information is:

- BLKSIZE=129
- LRECL=125
- RECFM=VBA

```
IBMBSTAB CSECT
ENTRY IBMBSTAB
IBMBSTAB BQU *
  DC 'IBMSTAB'
  DC H'14' OFFSET OF TAB COUNT
  DC H'60' PAGESIZE
  DC H'120' LINESIZE
  DC H'0' PAGELENGTH
  * (FOR TERMINALS)
  DC 3H'0' FILLERS
  * (RESERVED FOR
  * FUTURE USE)
  *
  DC H'5' TAB COUNT
  DC H'25' TAB 1
  DC H'49' TAB 2
  DC H'73' TAB 3
  DC H'97' TAB 4
  DC H'121' TAB 5
END
```

Figure 7-7. Tab control library module IBMBSTAB

**Example**

Figure 7-6 illustrates the use of a PRINT
file and the printing options of the
stream-oriented transmission statements to
format a table and write it onto magnetic
tape 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. After the last line of the
table has been written, the statement:

```
PUT FILE(TABLE) LINE(54)
```

causes the ENDPAGE condition to be raised
to ensure that a number appears at the foot
of the last page; the preceding statement
sets the flag FINISH to prevent a further
set of headings from being written.

The DD statement defining the data set
created by this program includes no record-
format information; the compiler infers the
following from the file declaration and the
line size specified in the statement that
opens the file TABLE:

- Record format = VBA (the default for a
  PRINT file)
- Record size = 98 (line size + one byte
  for printer control character + four bytes for
  record control field)
- Block size = 102 (record length + four
  bytes for block control field)

The program in Figure 8-10 uses record-
oriented transmission to print the table
created by the program in Figure 7-6.

**Tab Control Table**

Data-directed and list-directed output to a
PRINT file is automatically aligned on
preset tabulator positions; the tab
settings are stored in a table, an
assembler language control section, in the
transient library module IBMBSTAB (Figure
7-7.)

The standard settings are given in the
language reference manual for this
compiler. The functions of the fields in
the table are as follows:

- OFFSET OF TAB COUNT: Halfword binary integer that
defines the field that
  indicates the number of tabs
to be used.
- PAGESIZE: Halfword binary integer that
defines the default page
  size.
- LINESIZE: Halfword binary integer that
defines the default line size.
- PAGELENGTH: Halfword binary integer that
defines the default page
  length for printing at a
terminal. The page length is
the number of lines between
perforations. It is used for
DCL PLITABS STATIC EXT,
2 (OFFSET INIT(6),
PAGESIZE INIT(60),
LINESIZE INIT(120),
NO OF TABS INIT(3),
TAB1 INIT(30),
TAB2 INIT(60),
TAB3 INIT(90)) FIXED BIN(15,0);

Figure 7-8. PL/I structure PLITABS for modifying the standard tab settings (alternative method)

output to the terminal in a TSO environment because, unlike a printer, the terminal does not use a control tape to determine page length. The default value is zero which is a special convention to indicate unformatted output. For further information on how the output is formatted at a terminal, refer to the TSO Terminal User's Guide.

FILLERS: Reserved for future use.

Tab count: Number of tab position entries in table (maximum 255). If tab count = 0, any specified tab positions are ignored; each data item is positioned at the start of a new line.

Tab1-Tabn 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 will be ignored. The first data field in the printed output begins at the next available tab position.

The standard PL/I tab settings in IBMBSTAB can be overridden. If the linkage editor can resolve a reference to PLITABS generated by the compiler, the transient library module IBMBSTAB will not be used. Instead, the stream-oriented input/output routines will refer to the control section PLITABS for the tab settings.

There are two methods of altering the tab settings for a particular program.

One method is to create an assembler-language control section called PLITABS and include it in the link-editing of the program.

The alternative method is to include a PL/I structure in the source program. The organization of the structure is similar to the assembler-language control section for PLITABS given in Figure 7-7. The name of the structure must be PLITABS also and it must be declared STATIC and EXTERNAL. An example of a PL/I structure to create three tab settings in positions 30, 60, and 90, and use the defaults for page size and line size, is given in Figure 7-8.

The equivalent fields for PAGELENGTH and FILLERS are omitted from the structure, and the value given in the offset field is set to 6.

Note that the PAGESIZE field in PLITABS is used by PLIDUMP to define the pagesize for the dump output.

Standard Files

PL/I includes two standard files, SYSIN for input and SYSPRINT for output. If your program includes a GET statement that does not include the FILE option, the compiler inserts the file name SYSIN; if it includes a PUT statement without the FILE option, the compiler inserts the name SYSPRINT.

If you do not declare SYSPRINT, the compiler will give the file the attribute PRINT in addition to the normal default attributes; the complete file declaration will be:

SYSPRINT FILE STREAM OUTPUT PRINT EXTERNAL

Since SYSPRINT is a PRINT file, the compiler also supplies a default line size of 120 characters and a V-format record. You need give only a minimum of information in the corresponding DD statement; if your installation uses the usual convention that the system output device of class A is a printer, the following is sufficient:

//SYSPRINT DD SYSOUT=A

If you use one of the IBM-supplied cataloged procedures to execute your program, even this DD statement is not required, since it is included in the GO procedure step.

You can override the attributes given to SYSPRINT by the compiler by explicitly declaring or opening the file. If you do so, bear in mind that this file is also used by the error-handling routines of the compiler, and that any change you make in
the format of the output from SYSPRINT will also apply to the format of execution-time error messages. When an error message is printed, eight blanks are inserted at the start of each line except the first. If you specify a line size of less than 72 characters, the messages will not be output to SYSPRINT.

The compiler does not supply any special attributes for the standard input file SYSIN; if you do not declare it, it receives only the normal default attributes. The data set associated with SYSIN is usually in the input stream; if it is not in the input stream, you must supply full DD information.
This chapter describes how to define data sets for use with PL/I files that have the RECORD attribute. It explains how to create and access data sets for the three types of organization: CONSECUTIVE, INDEXED, and REGIONAL recognized by PL/I, and how to create and access data sets for teleprocessing. The essential parameters of the DD statements used in creating and accessing these data sets are summarized in tables, and several examples of PL/I programs (complete with JCL) are included to illustrate the text.

Data sets with the RECORD attribute are processed by record-oriented transmission in which data is transmitted to and from auxiliary storage exactly as it appears in the program variables; no data conversion takes place. A record in a data set corresponds to a variable in the program.

### Consecutive Data Sets

A data set with CONSECUTIVE organization can exist on any type of auxiliary storage device. Records are stored sequentially in the order in which you write them.

#### Creating a Consecutive Data Set

When you create a CONSECUTIVE data set you must specify:

- **Device that will write or punch your data set** (UNIT, SYSOUT, or VOLUME parameter of DD statement).
- **Block size**: you can specify the block size either in your PL/I program (ENVIRONMENT attribute) or in the DD statement (BLKSIZE subparameter). If you do not specify a record length, unblocked records are assumed and the record length is determined from the block size. If you do not specify a record format, U-format is assumed.

<table>
<thead>
<tr>
<th>Storage Device</th>
<th>Parameters of DD Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>All</td>
<td>When required</td>
</tr>
<tr>
<td>Direct access only</td>
<td>Always</td>
</tr>
<tr>
<td>Magnetic tape only</td>
<td>Always</td>
</tr>
<tr>
<td>Direct access and standard labeled magnetic tape</td>
<td>Data set not first in volume and for magnetic tapes that do not have standard labels</td>
</tr>
<tr>
<td>Data set to be kept after end of job</td>
<td>Disposition</td>
</tr>
<tr>
<td>Data set to be on particular device</td>
<td>Disposition</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Alternatively, you can specify the block size in your PL/I program by using the ENVIRONMENT attribute.

Figure 8-1. Creating a CONSECUTIVE data set: essential parameters of DD statement
If you want to keep a magnetic-tape or direct-access data set (that is, you do not want the operating system to delete it at the end of your job), the DD statement must name the data set and indicate how it is to be disposed of (DSNAME and DISP parameters). The DISP parameter alone will suffice if you want to use the data set in a later step but will not need it after the end of your job.

When creating a data set on a direct-access device, you must specify the amount of space required for it (SPACE parameter of DD statement).

If you want your data set stored on a particular magnetic-tape or direct-access device, you must indicate the volume serial number in the DD statement (SER or REF subparameter of VOLUME parameter).

If you do not supply a serial number for a magnetic-tape data set that you want to keep, the operating system will allocate one, inform the operator, and print the number on your program listing.

If your data set is to follow another data set on a magnetic-tape volume, you must use the LABEL parameter of the DD statement to indicate its sequence number on the tape. The essential information for creating a CONSECUTIVE data set is summarized in Figure 8-1.

The DCB subparameters of the DD statement that apply to CONSECUTIVE data sets are listed in Figure 8-2; they are described in Appendix A. You can specify record format (RECFM), block size (BLKSIZE), record size (LRECL), and number of buffers (BUFNO) in the ENVIRONMENT attribute of the DECLARE statement in your PL/I program instead of in a DD statement.

ACCESSING A CONSECUTIVE DATA SET

You can access a CONSECUTIVE data set in three ways. You can open the associated file for input, and read the records the data set contains; you can open the file for output, and extend the data set by adding records at the end; or you can open the file for update, and read and rewrite each record in turn. (The operating system does not permit updating a CONSECUTIVE data set on magnetic tape; you must read the data set and write the updated records into a new data set.)

To access a data set, you must identify it to the operating system in a DD statement. The following paragraphs, which are summarized in Figure 8-3, indicate the essential information you must include in the DD statement, and discuss some of the optional information you may supply. The discussions do not apply to data sets in the input stream, which are dealt with in Chapter 6.

<table>
<thead>
<tr>
<th>Subparameter</th>
<th>Specifies</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLKSIZE</td>
<td>Maximum number of bytes per block</td>
</tr>
<tr>
<td>BUFNO</td>
<td>Number of data management buffers</td>
</tr>
<tr>
<td>CODE</td>
<td>Paper tape: code in which the tape is punched</td>
</tr>
<tr>
<td>DEN</td>
<td>Magnetic tape: tape recording density</td>
</tr>
<tr>
<td>FUNC</td>
<td>Card reader or punch: function to be performed</td>
</tr>
<tr>
<td>LRECL</td>
<td>Maximum number of bytes per record</td>
</tr>
<tr>
<td>MODE</td>
<td>Card reader or punch: mode or operation (column binary or EBCDIC and read column eliminate or optical mark read)</td>
</tr>
<tr>
<td>OPTCD</td>
<td>Optional data-management services and data-set attributes</td>
</tr>
<tr>
<td>PRTSP</td>
<td>Printer line spacing (0, 1, 2, or 3)</td>
</tr>
<tr>
<td>RECFM</td>
<td>Record format and characteristics</td>
</tr>
<tr>
<td>STACK</td>
<td>Card reader or punch: stacker selection</td>
</tr>
<tr>
<td>TRTCH</td>
<td>Magnetic tape: tape recording technique for 7-track tape</td>
</tr>
</tbody>
</table>

Figure 8-2. DCB subparameters for CONSECUTIVE data sets

Essential Information

If the data set is cataloged, you need supply only the following information in the DD statement:

• The name of the data set (DSNAME parameter). The operating system will locate the information describing the data set in the system catalog, and, if
Parameters of DD Statement

- **When required**
  - When required
  - Always
  - If data set not cataloged

- **What you must state**
  - Name of data set
  - Disposition of data set
  - Input device
  - Standard labeled
  - Magnetic tape and volume serial number
  - Direct access
  - Sequence number
  - Block size
  - Magnetic tape: if data set first in volume or which does not have standard labels
  - If data set does not have standard labels

- **Parameters**
  - DSNAME=
  - DISP=
  - UNIT= or VOLUME=REF=
  - VOLUME=SER=
  - LABEL=
  - DCB=(BLKSIZE=...)

Figure 8-3. Accessing a CONSECUTIVE data set: essential parameters of DD statement

- Alternatively, you can specify the block size in your PL/I program by using either
  - the ENVIRONMENT attribute
  - the LINESIZE option.

necessary, will request the operator to mount the volume containing it.

- Confirmation that the data set exists (DISP parameter). If you open the data set for output with the intention of extending it by adding records at the end, code DISP=MOD; otherwise, to open the data set for output will result in its being overwritten.

If the data set is not cataloged, you must, in addition, specify the device that will read the data set and, for magnetic-tape and direct-access devices, give the serial number of the volume that contains the data set (UNIT and VOLUME parameters).

If the data set is on paper tape or punched cards, you must specify the block size either in your PL/I program (ENVIRONMENT attribute) or in the DD statement (BLKSIZE subparameter). The DSNAME parameter is not essential if the data set is not cataloged.

PL/I data management includes facilities for processing nonstandard labels which to the operating system appear as data sets preceding or following your data set. You can either process the labels as independent data sets or use the LABEL parameter of the DD statement to bypass them; to bypass the labels code LABEL=(2,NL) or LABEL=(,BLP).

**Record Format**

If you give record-format information, it must be compatible with the actual structure of the data set. For example, if a data set is created with F-format records, a record size of 600 bytes, and a block size of 3600 bytes, you can access the records as if they are U-format with a maximum block size of 3600 bytes; but if you specify a block size of 3500 bytes, your data will be truncated.

**Magnetic Tape Without Standard Labels**

If a magnetic-tape data set has nonstandard labels or is unlabeled, you must specify the block size either in your PL/I program (ENVIRONMENT attribute) or in the DD statement (BLKSIZE subparameter). The DSNAME parameter is not essential if the data set is not cataloged.

**EXAMPLE OF CONSECUTIVE DATA SETS**

Creating and accessing CONSECUTIVE data...
sets on magnetic tape are illustrated in the program of Figure 8-4. 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.

You cannot use a PRINT file for record-oriented transmission, and record-oriented transmission statements cannot include the printing options (PAGE, SKIP, etc). You can still exercise some control over the layout of printed output by including a printer control code as the first byte of each of your output records; you can also use similar control codes to select the stacker to which cards punched by your program are fed.
The operating system recognizes two types of code for printer and card punch commands, ANS code and machine code. You must indicate which code you are using, either in your PL/I program (ENVIRONMENT attribute), or in the DD statement (RECFM subparameter). If you specify one of these codes, but transmit your data to a device other than a printer or a card punch, the operating system will transmit the control bytes as part of your records. If you use an invalid control code, "space 1 line" or "stacker 1" will be assumed.

The ANS control codes, which are listed in Figure 8-5, cause the specified action to occur before the associated record is printed or punched.

<table>
<thead>
<tr>
<th>Code</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>Space 1 line before printing (blank code)</td>
</tr>
<tr>
<td>0</td>
<td>Space 2 lines before printing</td>
</tr>
<tr>
<td>-</td>
<td>Space 3 lines before printing</td>
</tr>
<tr>
<td>+</td>
<td>Suppress space before printing</td>
</tr>
<tr>
<td>1</td>
<td>Skip to channel 1</td>
</tr>
<tr>
<td>2</td>
<td>Skip to channel 2</td>
</tr>
<tr>
<td>3</td>
<td>Skip to channel 3</td>
</tr>
<tr>
<td>4</td>
<td>Skip to channel 4</td>
</tr>
<tr>
<td>5</td>
<td>Skip to channel 5</td>
</tr>
<tr>
<td>6</td>
<td>Skip to channel 6</td>
</tr>
<tr>
<td>7</td>
<td>Skip to channel 7</td>
</tr>
<tr>
<td>8</td>
<td>Skip to channel 8</td>
</tr>
<tr>
<td>9</td>
<td>Skip to channel 9</td>
</tr>
<tr>
<td>a</td>
<td>Skip to channel 10</td>
</tr>
<tr>
<td>b</td>
<td>Skip to channel 11</td>
</tr>
<tr>
<td>c</td>
<td>Skip to channel 12</td>
</tr>
<tr>
<td>v</td>
<td>Select stacker 1</td>
</tr>
<tr>
<td>W</td>
<td>Select stacker 2</td>
</tr>
</tbody>
</table>

The machine control codes differ according to the type of device. The codes for the 1403 Printer are listed in Figure 8.6, and Figure 8-7 gives those for the 2540 Card Read Punch. Control codes for the 3525 card printer are given in Figures 8-8 and 8-9.

The channel numbers refer to the printer carriage control tape. (See IBM 1403 Printer Component Description.)

Figure 8-5. ANS printer and card punch control codes
<table>
<thead>
<tr>
<th>Code byte</th>
<th>Print and then act</th>
<th>Action</th>
<th>Act immediately (no printing)</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000001</td>
<td>Print only (no space)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>00001001</td>
<td>Space 1 line</td>
<td>00010001</td>
<td>00010011</td>
</tr>
<tr>
<td>00010001</td>
<td>Space 2 lines</td>
<td>00010011</td>
<td>00010101</td>
</tr>
<tr>
<td>00011001</td>
<td>Space 3 lines</td>
<td>00011001</td>
<td>00011011</td>
</tr>
<tr>
<td>10001001</td>
<td>Skip to channel 1</td>
<td>10001001</td>
<td>10001011</td>
</tr>
<tr>
<td>10010001</td>
<td>Skip to channel 2</td>
<td>10010001</td>
<td>10010101</td>
</tr>
<tr>
<td>10011001</td>
<td>Skip to channel 3</td>
<td>10011001</td>
<td>10011011</td>
</tr>
<tr>
<td>10100001</td>
<td>Skip to channel 4</td>
<td>10100001</td>
<td>10100101</td>
</tr>
<tr>
<td>10101001</td>
<td>Skip to channel 5</td>
<td>10101001</td>
<td>10101101</td>
</tr>
<tr>
<td>10110001</td>
<td>Skip to channel 6</td>
<td>10110001</td>
<td>10110101</td>
</tr>
<tr>
<td>10111001</td>
<td>Skip to channel 7</td>
<td>10111001</td>
<td>10111101</td>
</tr>
<tr>
<td>11000001</td>
<td>Skip to channel 8</td>
<td>11000001</td>
<td>11001001</td>
</tr>
<tr>
<td>11001001</td>
<td>Skip to channel 9</td>
<td>11001001</td>
<td>11001101</td>
</tr>
<tr>
<td>11010001</td>
<td>Skip to channel 10</td>
<td>11010001</td>
<td>11011001</td>
</tr>
<tr>
<td>11011001</td>
<td>Skip to channel 11</td>
<td>11011001</td>
<td>11011101</td>
</tr>
<tr>
<td>11100001</td>
<td>Skip to channel 12</td>
<td>11100001</td>
<td>11101001</td>
</tr>
</tbody>
</table>

The channel numbers refer to the printer carriage control tape. (See IBM 1403 Printer Component Description.)

Figure 8-6. 1403 printer control codes

<table>
<thead>
<tr>
<th>Code byte</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000001</td>
<td>Select stacker 1</td>
</tr>
<tr>
<td>01000011</td>
<td>Select stacker 2</td>
</tr>
<tr>
<td>10000011</td>
<td>Select stacker 3</td>
</tr>
</tbody>
</table>

Figure 8-7. 2540 Card Read Punch control characters

<table>
<thead>
<tr>
<th>[CTLASA code]</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>Space 1 line and print</td>
</tr>
<tr>
<td>0</td>
<td>Space 2 lines and print</td>
</tr>
<tr>
<td>-</td>
<td>Space 3 lines and print</td>
</tr>
<tr>
<td>1</td>
<td>Skip to channel 1 and print</td>
</tr>
<tr>
<td>2</td>
<td>Skip to channel 2 and print</td>
</tr>
<tr>
<td>3</td>
<td>Skip to channel 3 and print</td>
</tr>
<tr>
<td>4</td>
<td>Skip to channel 4 and print</td>
</tr>
<tr>
<td>5</td>
<td>Skip to channel 5 and print</td>
</tr>
<tr>
<td>6</td>
<td>Skip to channel 6 and print</td>
</tr>
<tr>
<td>7</td>
<td>Skip to channel 7 and print</td>
</tr>
<tr>
<td>8</td>
<td>Skip to channel 8 and print</td>
</tr>
<tr>
<td>9</td>
<td>Skip to channel 9 and print</td>
</tr>
<tr>
<td>A</td>
<td>Skip to channel 10 and print</td>
</tr>
<tr>
<td>B</td>
<td>Skip to channel 11 and print</td>
</tr>
<tr>
<td>C</td>
<td>Skip to channel 12 and print</td>
</tr>
</tbody>
</table>

Figure 8-8. 3525 card printer control code (CTLASA)

<table>
<thead>
<tr>
<th>Code byte</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>00001001</td>
<td>Print on line 1</td>
</tr>
<tr>
<td>00010001</td>
<td>Print on line 2</td>
</tr>
<tr>
<td>00010101</td>
<td>Print on line 3</td>
</tr>
<tr>
<td>00011001</td>
<td>Print on line 4</td>
</tr>
<tr>
<td>00100011</td>
<td>Print on line 5</td>
</tr>
<tr>
<td>00101001</td>
<td>Print on line 6</td>
</tr>
<tr>
<td>00110011</td>
<td>Print on line 7</td>
</tr>
<tr>
<td>01000011</td>
<td>Print on line 8</td>
</tr>
<tr>
<td>01001001</td>
<td>Print on line 9</td>
</tr>
<tr>
<td>01010011</td>
<td>Print on line 10</td>
</tr>
<tr>
<td>01011001</td>
<td>Print on line 11</td>
</tr>
<tr>
<td>01100011</td>
<td>Print on line 12</td>
</tr>
<tr>
<td>01101001</td>
<td>Print on line 13</td>
</tr>
<tr>
<td>10000011</td>
<td>Print on line 14</td>
</tr>
<tr>
<td>10010001</td>
<td>Print on line 15</td>
</tr>
<tr>
<td>10010101</td>
<td>Print on line 16</td>
</tr>
<tr>
<td>10011001</td>
<td>Print on line 17</td>
</tr>
<tr>
<td>10100001</td>
<td>Print on line 18</td>
</tr>
<tr>
<td>10100101</td>
<td>Print on line 19</td>
</tr>
<tr>
<td>10101001</td>
<td>Print on line 20</td>
</tr>
<tr>
<td>10110001</td>
<td>Print on line 21</td>
</tr>
<tr>
<td>10110101</td>
<td>Print on line 22</td>
</tr>
<tr>
<td>11000001</td>
<td>Print on line 23</td>
</tr>
<tr>
<td>11001001</td>
<td>Print on line 24</td>
</tr>
<tr>
<td>11010001</td>
<td>Print on line 25</td>
</tr>
</tbody>
</table>

Figure 8-9. 3525 card printer control codes (CTL360)
There are two types of command 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 (you must include the second type of command 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 codes, 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 codes, include the control bytes 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 8-10 uses record-oriented transmission to read and print the contents of the data set SINES, created by the PRINT file in Figure 7-6. Since the data set SINES is cataloged, only two parameters are required in the DD statement that defines it. The output file PRINTER is declared with the ENVIRONMENT option CTLASA, specifying that the first byte of each record will be interpreted as an ANS printer control code. The information given in the ENVIRONMENT attribute could alternatively have been given in the DD statement, as follows:

```
DCB=(RECFM=VA,BLKSIZE=102)
```

**Indexed Data Sets**

A data set with INDEXED organization can exist only on a direct-access device. Each record in the data set is identified by a key that is recorded with the record. A key is a string of not more than 255 characters; all the keys in a data set must have the same length. The records in the data set are arranged according to the collating sequence of their keys. Once an INDEXED data set has been created, the keys facilitate the direct retrieval, addition, and deletion of records.

**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 keys of the last records on each track in the cylinder. A search can then be directed to the first track that has a key that is
higher than or equal to the key of the required record.

If the data set occupies more than one cylinder, the operating system develops a higher level index called a cylinder index. Each entry in the cylinder index identifies the key of the last record in the cylinder. To increase the speed of searching the cylinder index, you can request in a DD statement that the operating system develop a master index for a specified number of cylinders; you can have up to three levels of master index; Figure 8-11 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). Figure 8-12 shows how records are added to an INDEXED data set.

Each entry in the track index consists of two parts:

1. The normal entry, which points to the last record on the track.
2. 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.

CREATING AN INDEXED DATA SET

When you create an INDEXED data set, your program must write the records in the data set sequentially in the order of ascending key values; the associated file must be opened for SEQUENTIAL OUTPUT.
Figure 8-12. Adding records to an INDEXED data set
You can use a single DD statement to define the whole of the data set (index area, prime area, and overflow area), or you can use two or three statements to define the areas independently. If you use two DD statements, you can define either the index area and the prime area together, or the prime area and the overflow area together.

If you want the whole of the data set to be on a single volume, there is no advantage to be gained by using more than one DD statement except to define an independent overflow area (see "overflow Area," later in this chapter). But, if you use separate DD statements to define the index and/or overflow area on volumes separate from that which contains the prime area, you will increase the speed of direct access to the records in the data set by reducing the number of access mechanism movements required.

When you use two or three DD statements to define an INDEXED data set, the statements must appear in the order: index area; prime area; overflow area. The DD statement must have a name (ddname) but the name fields of a second or third DD statement must be blank. The DD statements for the prime and overflow areas must specify the same type of unit (UNIT parameter). You must include all the DCB information for the data set in the first DD statement; DCB=DSORG=IS will suffice in the other statements.

An INDEXED data set consisting of fixed-length records can be extended by adding records sequentially at the end, until the original space allocated for the prime data is filled. The corresponding file must be opened for sequential output and you must include DISP=MOD in the DD statement.

**Essential Information**

To create an INDEXED data set, you must give the operating system certain information either in your PL/I program or in the DD statement that defines the data set. The following paragraphs indicate the essential information, and discuss some of the optional information you may supply; the ENVIRONMENT attribute and the LINESIZE option are discussed fully in the language reference manual for this compiler.

You must supply the following information when creating an INDEXED data set:

- Device that will write or punch your data set (UNIT or VOLUME parameter of DD statement).
- Block size: you can specify the block size either in your PL/I program (ENVIRONMENT attribute or LINESIZE option) or in the DD statement (ELKSIZE subparameter). If you do not specify a record length, unblocked records are assumed and the record length is determined from the block size.

If you want to keep a direct-access data set (that is, you do not want the operating system to delete it at the end of your job), the DD statement must name the data set and indicate how it is to be disposed of (DSNAME and DISP parameters). The DISP parameter alone will suffice if you want to use the data set in a later step but will not need it after the end of your job.

If you want your data set stored on a particular direct-access device, you must indicate the volume serial number in the DD statement (SER or REF subparameter of VOLUME parameter). If you do not supply a serial number for a data set that you want to keep, the operating system will allocate one, inform the operator, and print the number on your program listing. All the essential parameters required in a DD statement for the creation of an INDEXED data set are summarized in Figure 8-13, and Figure 8-14 lists the DCB subparameters needed.

Appendix A contains a description of the DCB subparameters.

You cannot place an INDEXED data set on a system output (SYSOUT) device.

You must request space for the prime data area in the SPACE parameter. Your request must be in units of cylinders unless you place the data set in a specific position on the volume (by specifying a track number in the SPACE parameter). In the latter case, the number of tracks you specify must be equivalent to an integral number of cylinders, and the first track must be the first track of a cylinder other than the first cylinder in the volume. You can also use the SPACE parameter to specify the amount of space to be used for the cylinder and master indexes (unless you use a separate DD statement for this purpose). If you do not specify the space for the indexes, the operating system will use part of the independent overflow area; if there is no independent overflow area, it will use part of the prime data area.

In the DCB parameter, you must always specify the data set organization (DSORG=IS), and in the first (or only) DD statement you must also specify the length of the key (KEYLEN).
Parameters of DD Statement

<table>
<thead>
<tr>
<th>When required</th>
<th>What you must state</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output device</td>
<td>UNIT= or VOLUME=REF=</td>
<td></td>
</tr>
<tr>
<td>Storage space required</td>
<td>SPACE=</td>
<td></td>
</tr>
<tr>
<td>Data control block</td>
<td>DCB=</td>
<td></td>
</tr>
<tr>
<td>More than one DD statement</td>
<td>Name of data set and</td>
<td>DSNAME=</td>
</tr>
<tr>
<td>Data set to be used in another job</td>
<td>Disposition</td>
<td>DISP=</td>
</tr>
<tr>
<td>Data set to be kept after end of</td>
<td>Disposition</td>
<td>DISP=</td>
</tr>
<tr>
<td>job</td>
<td>Name of data set</td>
<td>DSNAME=</td>
</tr>
<tr>
<td>Data set to be on particular</td>
<td>Volume serial number</td>
<td>VOLUME=SER=</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VOLUME=REF=</td>
</tr>
</tbody>
</table>

Figure 8-13. Creating an INDEXED data set: essential parameters of DD statement

Name of Data Set

If you use only one DD statement to define your data set, you need not name the data set unless you intend to access it in another job. But, if you include two or three DD statements, you must specify a data set name, even for a temporary data set.

The DSNAME parameter in a DD statement that defines an INDEXED data set not only gives the data set a name, but it also identifies the area of the data set to which the DD statement refers:

- DSNAME=name(INDEX)
- DSNAME=name(PRIME)
- DSNAME=name(OVFLOW)

If the data set is temporary, prefix its name with the characters "$$. If you use one DD statement to define the prime and index or prime and overflow area, code DSNAME=name(PRIME); if you use one DD statement, code DSNAME=name(PRIME), or simply DSNAME=name.

Record Format and Keys

An INDEXED data set can contain either fixed-length or variable-length records, blocked or unblocked. You must always specify the record format either in your PL/I program (ENVIRONMENT attribute) or in the DD statement (RECFM subparameter).

The key associated with each record can be contiguous with or embedded within the data in the record; you can save storage space in the data set if you use blocked records with embedded keys.

If the records are unblocked, the key of each record is recorded in the data set in front of the record even if it is also embedded within the record, as shown in (a) and (b) of Figure 8-15. If blocked records do not have embedded keys, the key of each record is recorded within the block in front of the record, and the key of the last record in the block is also recorded in front of the block, as shown in (c) of Figure 8-15. When blocked records have embedded keys, the individual keys are not recorded separately in front of each record in the block; the key of the last record in the block is recorded in front of the block, as shown in (d) of Figure 8-15.

If you use blocked records with non-
### DCB Subparameters

<table>
<thead>
<tr>
<th>When required</th>
<th>To specify</th>
<th>Subparameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Record format</td>
<td>RECFM=F, FB, FBS, V, or VB</td>
<td></td>
</tr>
<tr>
<td>Block size</td>
<td>BLKSIZE=</td>
<td></td>
</tr>
<tr>
<td>Data set organization</td>
<td>DSORG=IS</td>
<td></td>
</tr>
<tr>
<td>Key length</td>
<td>KEYLEN=</td>
<td></td>
</tr>
<tr>
<td>Cylinder overflow area and number of tracks per cylinder for overflow records</td>
<td>CYLOFL=</td>
<td></td>
</tr>
<tr>
<td>Independent overflow area</td>
<td>OPTCD=I</td>
<td></td>
</tr>
<tr>
<td>Record length</td>
<td>LRECL=</td>
<td></td>
</tr>
<tr>
<td>Embedded key (relative key position)</td>
<td>RKP=</td>
<td></td>
</tr>
<tr>
<td>Master index</td>
<td>OPTCD=M</td>
<td></td>
</tr>
<tr>
<td>Automatic processing of dummy records</td>
<td>OPTCD=L</td>
<td></td>
</tr>
<tr>
<td>Number of data management buffers</td>
<td>BUFNO=</td>
<td></td>
</tr>
<tr>
<td>Number of tracks in cylinder index</td>
<td>NTM=</td>
<td></td>
</tr>
<tr>
<td>for each master index entry</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**These are always required**

**These are optional**

---

1. Alternatively, can be specified in ENVIRONMENT attribute.

Note: Full DCB information must appear in the first, or only, DD statement. Subsequent statements require only DSORG=IS.

**Figure 8-14. DCB subparameters for an INDEXED data set**

Embedded keys, the record size that you specify must include the length of the key, and the block size must be a multiple of this combined length. Otherwise, record length and block size refer only to the data in the record. Record format information is shown in Figure 8-16.

If you use records with embedded keys, you must include the DCB subparameter RKP to indicate the position of the key within the record. For fixed-length records the value specified in the RKP subparameter is one less than the byte number of the first character of the key; that is, if RKP=1, the key starts in the second byte of the record. The value assumed if you omit this subparameter is RKP=0, which specifies that the key is not embedded in the record but is separate from it.

For variable-length records, the value specified in the RKP subparameter must be the relative position of the key within the record plus four. The extra four bytes take into account the 4-byte control field used with variable-length records. For this reason you must never specify RKP less than four. When deleting records you must always specify RKP equal to or greater than five, since the first byte of the data is used to indicate deletion.

For unblocked records, the key, even if embedded, is always recorded in a position preceding the actual data. Consequently, the RKP subparameter need not be specified for unblocked records.

**Overflow Area**

If you intend to add records to the data set on a future occasion, you must request either a cylinder overflow area or an independent overflow area, or both.

For a cylinder overflow area, include...
(a) Unblocked records, non-embedded keys

| Key | Data | Key | Data | Key | Data |

(b) Unblocked records, embedded keys

| Key | Key | Data | Key | Key | Data | Key | Key |

| Key | Key | Data |

(c) Blocked records, non-embedded keys

| Key | Key | Data | Key | Data | Key | Data | Key | Key |

(d) Blocked records, embedded keys

| Key | Key | Key | Key | Data |

(e) Unblocked variable length records, \( RKP > 4 \)

| Key | B1 | R1 | Key | Data |

| Key | B1 | R1 | Key |

(f) Blocked variable length records, \( RKP > 4 \)

| Key | B1 | R1 | Key | R1 | Key | R1 | Key |

| Key | B1 | R1 | Key |

(g) Unblocked variable length records, \( RKP = 4 \)

| Key | B1 | R1 | Key |

(h) Blocked variable length records, \( RKP = 4 \)

| Key | B1 | R1 | Key | R1 | Key | R1 | Key |

| Key | B1 | R1 | Key |

Figure 8-15. Record formats in an INDEXED data set
Not zero | R | R*B
---|---|---

<table>
<thead>
<tr>
<th>Blocked records</th>
<th>Zero or</th>
<th>R + K</th>
<th>B*(R+K)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>omitted</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Unblocked records</th>
<th>Not zero</th>
<th>R</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>omitted</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

R = Size of data in record
K = Length of keys (as specified in KEYLEN subparameter)
B = Blocking factor

Example: For blocked records,
non-embedded keys, 100 bytes of data per record, 10 records per block, key length = 20:
LRECL=120,BLKSIZ=1200,RECFM=FB

Figure 8-16. Record format information for an INDEXED data set

The DCB subparameter OPTCD=Y and use the subparameter CYLOFL to specify the number of tracks in each cylinder to be reserved for overflow records. A cylinder overflow area has the advantage of a short search time for overflow records, but the amount of space available for overflow records is limited, and much of the space may be unused if the overflow records are not evenly distributed throughout the data set.

For an independent overflow area, use the DCB subparameter OPTCD=I to indicate that overflow records are to be placed in an area reserved for overflow records from all cylinders, and include a separate DD statement to define the overflow area. The use of an independent area has the advantage of reducing the amount of unused space for overflow records, but entails an increased search time for overflow records.

It is good practice to request cylinder overflow areas large enough to contain a reasonable number of additional records and an independent overflow area to be used as the cylinder overflow areas are filled.

If the prime data area is not filled during creation, you cannot use the unused portion for overflow records, nor for any records subsequently added during direct access (although you can fill the unfilled portion of the last track used). You can reserve space for later use within the prime data area by writing "dummy" records during creation: see "Dummy Records," later in this chapter.

Master Index

If you want the operating system to create a master index for you, include the DCB subparameter OPTCD=M, and indicate in the NTM subparameter the number of tracks in the cylinder index you wish to be referred to by each entry in the master index. The operating system will automatically create up to three levels of master index, the first two levels addressing tracks in the next lower level of the master index.

Dummy 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, and you must decide whether you want the operating system to identify and skip dummy (deleted) records.

If you code OPTCD=L, the operating system will identify any record that is named in a DELETE statement by placing the bit string (8)'1'B in the first byte. Subsequently, during SEQUENTIAL processing of the data set, such records will be ignored; if they are forced off a track when the data set is being updated, they will not be placed in the overflow area. Do not specify OPTCD=L when using blocked or variable-length records with non-embedded keys; if you do, the string (8)'1'B will overwrite the key of the "deleted" record.

You can include a dummy record in an INDEXED data set by setting the first byte of data to (8)'1'B and writing the record in the usual way.

Accessing an Indexed Data Set

You can open an INDEXED data set for sequential or direct access, and for input or update in each case. 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; during sequential access, if OPTCD=L is specified when the data set is created, dummy records are ignored. Using direct
### Parameters of DD Statement

<table>
<thead>
<tr>
<th>When required</th>
<th>What you must state</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Always</td>
<td>Name of data set</td>
<td>DSNAME=</td>
</tr>
<tr>
<td></td>
<td>Disposition of data set</td>
<td>DISP=</td>
</tr>
<tr>
<td></td>
<td>Data control block information</td>
<td>DCB=</td>
</tr>
<tr>
<td>If data set not cataloged</td>
<td>Input device</td>
<td>UNIT= or VOLUME=REF=</td>
</tr>
<tr>
<td></td>
<td>Volume serial number</td>
<td>VOLUME=SER=</td>
</tr>
</tbody>
</table>

Figure 8-17. Accessing an INDEXED data set: essential parameters of DD statement

To access an INDEXED data set, you must define it in one, two or three DD statements; the DD statements must correspond with those used when the data set is created. The following paragraphs indicate the essential information you must include in each DD statement. Figure 8-17 summarizes this information.

If the data set is cataloged, you need supply only the following information in each DD statement:

- The name of the data set (DSNAME parameter). The operating system will locate the information that describes the data set in the system catalog and, if necessary, will request the operator to mount the volume that contains it.
- Confirmation that the data set exists (DISP parameter).
- Full DCB information for the first, or only, DD statement. Subsequent statements require only DSORG=IS to be coded.

If the data set is not cataloged, you must, in addition, specify the device that will process the data set and give the serial number of the volume that contains it (UNIT and VOLUME parameters).

#### Reorganizing an Indexed Data Set

It is necessary to reorganize an INDEXED data set periodically because the addition of records to the data set results in an increasing number of records in the overflow area. Therefore, even if the overflow area does not eventually become full, the average time required for the direct retrieval of a record will increase. The frequency of reorganization depends on how often the data set is updated, on how much storage is available in the data set, and on your timing requirements.

Reorganizing the data set also eliminates records that are marked as "deleted," but are still present within the data set.

There are two ways to reorganize an INDEXED data set:

1. Read the data set into an area of main storage or onto a temporary CONSECUTIVE data set, and then recreate it in the original area of auxiliary storage.
2. Read the data set sequentially and write it into a new area of auxiliary storage; you can then release the original auxiliary storage.

#### Examples of Indexed Data Sets

The creation of a simple INDEXED data set is illustrated in Figure 8-18. The data set contains a telephone directory, using the subscribers' names as keys to the telephone numbers.

The program in Figure 8-19 updates this data set and prints out its new contents. The input data includes the following codes to indicate the operations required:
DCL DIREC FILE RECORD SEQUENTIAL KEYED ENV(INDEXED),
CARD CHAR(80),
NAME CHAR(20) DEF CARD POS(1),
NUMBER CHAR(3) DEF CARD POS(21),
IOFIELD CHAR(3);

ON ENDFILE(SYSIN) GO TO FINISH;

OPEN FILE(DIREC) OUTPUT;
NEXTIN: GET FILE(SYSIN) EDIT(CARD) (A(80));
IOFIELD=NUMBER;
WRITE FILE(DIREC) FROM(IOFIELD) KEYFROM(NAME);
END TO NEXTIN;
FINISH: CLOSE FILE(DIREC);
END TELNOS;

//GO.DIREC DD DSNAME=TELNO(INDEX),UNIT=2311,SPACE=(CYL,1),
// DCB=(RECFM=F,BLKSIZE=3,DSORG=IS,KEYLEN=20,OPTCD=LIY,CYLOFL=2),
// DISP=(NEW,KEEP),VOLUME=SER=D186
// DD DSNAME=TELNO(PRIME),UNIT=2311,SPACE=(CYL,4),DCB=DSORG=IS,
// DISP=(NEW,KEEP),VOLUME=SER=D186
// DD DSNAME=TELNO(OVFLOW),UNIT=2311,SPACE=(CYL,4),
// DCB=DSORG=IS,DISP=(NEW,KEEP),VOLUME=SER=D186

Figure 8-18. Creating an INDEXED data set

A: Add a new record
C: Change an existing record
D: Delete an existing record

Regional Data Sets

A data set with REGIONAL organization can exist only on a direct-access device. A REGIONAL data set is divided into regions that are numbered consecutively from zero. The following paragraphs briefly describe the three types of REGIONAL organization.

REGIONAL(1): In this organization a region is a record. Each record in the data set is identified by its region number, an unsigned decimal integer not exceeding 16777215. Region numbers start from 0 at the beginning of the data set.
DCL DIREC FILE RECORD KEYED ENV(INDEXED),
ONCODE BUILTIN,
NUMBER CHAR(3),
NAME CHAR(20),
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 (NUMBER) KEY FROM (NAME);
   ELSE IF CODE='C' THEN REWRITE FILE(DIREC) FROM (NUMBER)
   KEY (NAME);
   ELSE IF CODE='D' THEN DELETE FILE(DIREC) KEY (NAME);
   ELSE PUT FILE(SYSPRINT) SKIP EDIT ('INVALID CODE:',NAME)
   (A(15),A);
GO TO NEXT;
PRINT: CLOSE FILE(DIREC);
PUT FILE(SYSPRINT) PAGE;
OPEN FILE(DIREC) SEQUENTIAL INPUT;
ON ENDFILE(DIREC) GO TO FINISH;
NEXTIN: READ FILE(DIREC) INTO (NUMBER) KEY TO (NAME);
   PUT FILE(SYSPRINT) SKIP EDIT (NAME,NUMBER) (A);
GO TO NEXTIN;
FINISH: CLOSE FILE(DIREC);
END DIRUPDT;
/*
//GO.DIREC DD DSN=TELNO(INDEX),UNIT=2311,VOL=SER=D186,DISP=(OLD,KEEP)
// DD DSN=TELNO(PRIME),UNIT=2311,VOL=SER=D186,DISP=(OLD,KEEP)
// DD DSN=TELNO(OVFLOW),UNIT=2311,VOL=SER=D186,DISP=(OLD,KEEP)
//GO.SYSIN DD *
NEWMAN,M.W. 516 C
GOODFELLOW,D.T. 889 A
MILES,R. 209 D
HARVEY,C.D.W. 183 A
BARTLETT,S.G. 001 D
CORY,G. 439 A
READ,K.M. PITT,W.H. 291 A
CHALLENGE, D.F. 001 A
HASTINGS, G.M. 439 D
BRAMLEY,O.H. 439 C
/*
Figure 8-19. Updating an INDEXED data set

Chapter 8: Defining Data Sets for Record Files 117
REGIONAL(1) data sets have no recorded keys. Note, however, that PL/I REGIONAL(1) DIRECT INPUT or UPDATE files can be used to process data sets that do have recorded keys. In particular, REGIONAL(2) and REGIONAL(3) data sets can be accessed by a file declared as ENV(REGIONAL(1)).

REGIONAL(2): This organization is similar to REGIONAL(1), but differs, in that a key is recorded with each record. The recorded key is a string of not more than 255 characters. For files with the DIRECT attribute, a record is written in the first vacant space on the track that contains the region number specified in the WRITE statement; for retrieval, the search for a record begins on the track that contains the region number specified in the READ statement, and may continue through the data set until the record has been found. For files that are created sequentially, the record is written sequentially.

REGIONAL(3): This organization is similar to REGIONAL(2), but differs in that each region corresponds to one track of the direct-access device and is not a record position. Depending on the record length, a region can contain one or more records.

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. REGIONAL(1) organization is most suited to applications where there will be no duplicate region numbers, and where most of the regions will be filled (reducing wasted space in the data set). REGIONAL(2) and REGIONAL(3) are more appropriate where records are identified by numbers that are thinly distributed over a wide range. You can include in your program an algorithm that derives the region number from the number that identifies a record in such a manner as to optimize the use of space within the data set; duplicate region numbers may occur but, unless they are on the same track, their only effect might be to lengthen the search time for records with duplicate region numbers.

REGIONAL(1) and REGIONAL(2) data sets can contain only F-format unblocked records, but a REGIONAL(3) data set can have unblocked records of all three formats, F, V, and U. The examples at the end of this section illustrate typical applications of all three types of REGIONAL organization.

Creating a Regional Data Set

You can use either sequential or direct-access to create a REGIONAL data set.

In sequential creation, you must present records in order of ascending region numbers; for REGIONAL(1) and REGIONAL(2) the region number for each record must exceed that of the preceding record since each region can contain only one record. In all cases, dummy records (identified by (8)'1'B in the first byte) are placed automatically in regions whose numbers are skipped. The data set can have up to 15 extents, which may be on more than one volume.

For direct creation, one of the PL/I library subroutines formats the whole of the data set when you open the corresponding file. For REGIONAL(1) and (2), and for REGIONAL(3) with F-format records, formatting involves filling the data set with dummy records; for REGIONAL(3) with U-format or V-format records, a record, called the capacity record is written at the start of each track to indicate an empty track. During creation, you can present records in any order. The data set can have only one extent, and can therefore reside on only one volume.

Essential Information

To create a REGIONAL data set, you must give the operating system certain information either in your PL/I program or in the DD statement that defines the data set. The following paragraphs indicate the essential information, and discuss some of the optional information you may supply; the ENVIRONMENT attribute and the LINESIZE option are discussed fully in the language reference manual for this compiler.

You must supply the following information when creating a REGIONAL data set:

- Device that will write or punch your data set (UNIT or VOLUME parameter of DD statement).

- Block size: you can specify the block size either in your PL/I program (ENVIRONMENT attribute or LINESIZE option) or in the DD statement (BLKSIZE subparameter). If you do not specify a record length, unblocked records are assumed and the record length is determined from the block size.
### Parameters of DD Statement

<table>
<thead>
<tr>
<th>When required</th>
<th>What you must state</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output device</td>
<td>UNIT= or VOLUME=REF=</td>
<td></td>
</tr>
<tr>
<td>Storage space required</td>
<td>SPACE=</td>
<td></td>
</tr>
<tr>
<td>Data control block information: refer to figure 8-21</td>
<td>DCB=</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data set to be used in another job but not required in another job</th>
<th>Disposition</th>
<th>DISP=</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data set to be kept after end of job</td>
<td>Disposition</td>
<td>DISP=</td>
</tr>
<tr>
<td>Name of data set</td>
<td>DSNAME=</td>
<td></td>
</tr>
<tr>
<td>Data set to be on particular volume</td>
<td>VOLUME=SER=</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VOLUME=REF=</td>
<td></td>
</tr>
</tbody>
</table>

Figure 8-20. Creating a REGIONAL data set: essential parameters of DD statement

### DCB Subparameters

<table>
<thead>
<tr>
<th>When required</th>
<th>To specify</th>
<th>Subparameters</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Record format(^1)</td>
<td>REC(F)=</td>
</tr>
<tr>
<td>These are always required</td>
<td>(\text{or REC}F)=V(^2) REGIONAL(3) only (\text{or REC}F)=U \text{REGIONAL(3) only}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Block size(^4)</td>
<td>BLK(\text{SIZE})=</td>
</tr>
<tr>
<td></td>
<td>Data set organization</td>
<td>DS(\text{ORG})=DA</td>
</tr>
<tr>
<td></td>
<td>Key length (REGIONAL(2) and (3) only)</td>
<td>KEY(\text{LEN})=</td>
</tr>
<tr>
<td></td>
<td>Limited search for a record or space to add a record (REGIONAL(2) and (3) only)</td>
<td>LIM(\text{CT})=</td>
</tr>
<tr>
<td>These are optional</td>
<td>Number of data management buffers(^1)</td>
<td>BUF(\text{NO})=</td>
</tr>
</tbody>
</table>

\(^1\) Alternatively, can be specified in ENVIRONMENT attribute.
\(^2\) REC\(FM\)=VS must be specified in the ENVIRONMENT attribute for sequential input or update.

Figure 8-21. DCB subparameters for a REGIONAL data set
If you want to keep a data set (that is, you do not want the operating system to delete it at the end of your job), the DD statement must name the data set and indicate how it is to be disposed of (DSNAME and DISP parameters). The DISP parameter alone will suffice if you want to use the data set in a later step but will not need it after the end of your job.

If you want your data set stored on a particular direct-access device, you must indicate the volume serial number in the DD statement (SER or REF subparameter of VOLUME parameter). If you do not supply a serial number for a data set that you want to keep, the operating system will allocate one, inform the operator, and print the number on your program listing. All the essential parameters required in a DD statement for the creation of a REGIONAL data set are summarized in Figure 8-20, and Figure 8-21 lists the DCB subparameters needed. Appendix A contains a description of the DCB subparameters.

You cannot place a REGIONAL data set on a system output (SYSOUT) device.

In the DCB parameter, you must always specify the data set organization as direct by coding DSORG=DA. For REGIONAL(2) and REGIONAL(3), you must also specify the length of the recorded key (KEYLEN); refer to the language reference manual for this compiler for a description of how the recorded key is derived from the source key supplied in the KEYFROM option.

For REGIONAL(2) and REGIONAL(3), if you want to restrict the search for space to add a new record, or the search for an existing record, to a limited number of tracks beyond the track that contains the specified region, use the LIMCT subparameter of the DCB parameter. If you omit this parameter, the search will continue to the end of the data set, and then from the beginning of the data set back to the starting point in the data set.

ACCESSING A REGIONAL DATA SET

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 may rewrite each record in turn. Sequential access of a REGIONAL(2) or REGIONAL(3) data set will give you the records in the order in which they appear in the data set, which is not necessarily region number order.

Using direct input, you can read any record by supplying its region number and, for REGIONAL(2) and REGIONAL(3), its recorded key; in direct update, you can read or delete existing records or add new ones. The operating system ignores dummy records in a REGIONAL(2) or REGIONAL(3) data set; but a program that processes a REGIONAL(1) data set must be prepared to recognize dummy records.

To access a REGIONAL data set, you must identify it to the operating system in a DD statement. The following paragraphs indicate the minimum information you must include in the DD statement; this information is summarized in Figure 8-22.

If the data set is cataloged, you need only supply the following information in your DD statement:

- The name of the data set (DSNAME parameter). The operating system will locate the information that describes the data set in the system catalog and, if necessary, will request the operator to mount the volume that contains it.
- Confirmation that the data set exists (DISP parameter).

If the data set is not cataloged, you must, in addition, specify the device that will read the data set and give the serial number of the volume that contains the data set (UNIT and VOLUME parameters).

EXAMPLES OF REGIONAL DATA SETS

REGIONAL(1) Data Sets

Creating a REGIONAL(1) data set is illustrated in Figure 8-23.

The program uses the same data as that in Figure 8-18, 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 with the region numbers in the data set, the data in each occupied region being a subscriber's name. The SPACE parameter of the DD statement requests space for 1000 twenty-byte records (that is, for 1000 regions); since space is never allocated in units of less than one track and one 2311 track can accommodate 45 twenty-byte records, there will in fact be 1035 regions. Note that the data set has no recorded keys because it is created using a DIRECT OUTPUT file.
Illustrated in Figure 8-24. The data read interpreted in the same way. Like the program in Figure 8-19, this program data set, each record is tested and dummy in a REGIONAL(1) data set even if it is not dummy; this is necessary because a WRITE statement can overwrite an existing record in the region is tested to ensure that it is a dummy; this is necessary because a WRITE statement can overwrite an existing record in a REGIONAL(1) data set even if it is not a dummy. Similarly, during the sequential reading and printing of the contents of the data set, each record is tested and dummy records are not printed.

The use of REGIONAL(2) data sets is illustrated in Figure 8-25, Figure 8-26, and Figure 8-27. The programs in these figures perform the same functions as those given for REGIONAL(3), with which they can usefully be compared.

The programs depict a library processing scheme, in which loans of books are recorded and reminders are issued for overdue books. Two data sets, STOCK2 and LOANS2, are involved. STOCK2 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.) LOANS2 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 LOANS2; the reader and book numbers are concatenated to form the recorded keys.

In Figure 8-25, the data sets STOCK2 and LOANS2 are created. The file LOANS, which is used to create the data set LOANS2 is opened for direct output merely to format the data set; the file is closed immediately 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 parameter in the DD statement that defines LOANS2 requests 100 blocks of 19 bytes (12 bytes of data and a 7-byte key; see Figure 8-26). Direct creation is also used for the data set STOCK2 because, even if the input data is presented in ascending reference number order, identical region numbers might be derived from successive reference numbers.

Updating of the data set LOANS2 is illustrated in Figure 8-26. 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 date. This date is written in both the issue date and reminder date portions of a new record or an updated record.

The program in Figure 8-27 uses a sequential update file (LOANS) to process the records in the data set LOANS2, and a direct input file (STOCK) to obtain the book description from the data set STOCK2 for use in a reminder note. Each record from LOANS is tested to see whether the last reminder was issued more than a month ago; if necessary, a reminder note is issued and the current date is written in the reminder date field of the record.

Parameters of DD Statement

<table>
<thead>
<tr>
<th>When required</th>
<th>What you must state</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Always</td>
<td>Name of data set</td>
<td>DSNAME=</td>
</tr>
<tr>
<td></td>
<td>Disposition of data set</td>
<td>DISP=</td>
</tr>
<tr>
<td>If data set not cataloged</td>
<td>Input device</td>
<td>UNIT= or VOLUME=REF=</td>
</tr>
<tr>
<td></td>
<td>Volume serial number</td>
<td>VOLUME=SER=</td>
</tr>
</tbody>
</table>

Figure 8-22. Accessing a REGIONAL data set: essential parameters of DD statement

Chapter 8: Defining Data Sets for Record Files 121
//OPTS#23 JOB
//STEP1 EXEC PLIXCLG
//PLI.SYSIN DD *
CRR1: PROC OPTIONS(MAIN);

DCL NOS FILE RECORD OUTPUT DIRECT KEYED ENV(REGIONAL(1)),
    CARD CHAR(80),
    NAME CHAR(20) DEF CARD POS(1),
    NUMBER CHAR(3) DEF CARD POS(21),
    IOFIELD CHAR(20);

ON ENDFILE (SYSIN) GO TO FINISH;

NEXT: GET FILE (SYSIN) EDIT (CARD) (A(80));
    IOFIELD=NAME;
    WRITE FILE (NOS) FROM (IOFIELD) KEY FROM (NUMBER);
    GO TO NEXT;
FINISH: CLOSE FILE (NOS);
END CRR1;

Figure 8-23. Creating a REGIONAL(1) data set

REGIONAL(3) Data Sets

The use of REGIONAL(3) data sets, illustrated in Figure 8-28, Figure 8-29, and Figure 8-30, is similar to the REGIONAL(2) figures, above; only the important differences are discussed here.

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. The average record (including the recorded key) is assumed to be 60 bytes; therefore the average number of records per track (that is, per region) is 25, and there will be 40 regions.

In Figure 8-28, the data set STOCK3 is created sequentially; duplicate region numbers are acceptable, because each region can contain more than one record.

In Figure 8-29, the region number for the data set LOANS3 is obtained simply by testing the reader number; there are only three regions, because a 2311 track can hold 36 nineteen-byte records.

The only notable difference between Figure 8-30 and the corresponding
//OPT8#24 JOB
//STEP1 EXEC PLXCLG
//PLI.SYSIN DD *
ACRI: PROC OPTIONS(MAIN);

DCL NOS FILE RECORD KEYED ENV(REGIONAL(1)),
  NAME CHAR(20),
  (NEWNO,OLDNO) CHAR(3),
  CODE CHAR(1),
  IOFIELD CHAR(20),
  BYTE1 CHAR(1) DEF IOFIELD POS(1);

ON ENDFILE(SYSIN) GO TO PRINT;

OPEN FILE(NOS) DIRECT UPDATE;

NEXT:
  GET FILE(SYSIN) EDIT(NAME,NEWNO,OLDNO,CODE)
    (A(20),2 A(3),X(53),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:
  READ FILE(NOS) KEY(NEWNO) INTO(IOFIELD);
  IF UNSPEC(BYTE1)=(8)'1' THEN WRITE FILE(NOS) KEYFROM(NEWNO)
    FROM NAME;
  ELSE PUT FILE(SYSPRINT) SKIP EDIT('DUPLICATE:',NAME) (A(15),A);
  GO TO NEXT;

PRINT:
  CLOSE FILE(NOS);
  PUT FILE(SYSPRINT) PAGE;
  OPEN FILE(NOS) SEQUENTIAL INPUT;

ON ENDFILE(NOS) GO TO FINISH;

NEXTN:
  READ FILE(NOS) INTO(IOFIELD) KEYTO(NEWNO);
  IF UNSPEC(BYTE1)=(8)'1' THEN GO TO NEXTN;
  ELSE PUT FILE(SYSPRINT) SKIP EDIT(NEWNO,IOFIELD) (A(5),A);
  GO TO NEXT;

FINISH:
  CLOSE FILE(NOS);
END ACRI;

/*
//GO.NOS DD DSN=NOSA,UNIT=2311,VOL=SER=D186,DISP=(GLD,KEEP)
//GO.SYSIN DD *
NEWMAN,M.W  516450  C
GOODFELLOW,D.T.  889  A
MILES,R.  233  D
HARVEY,C.D.W.  209  A
BARTLETT,S.G.  183  A
CORY,G.  336  D
READ,K.M.  001  A
PITT,W.H.  515  A
ROLF,D.F.  114  D
ELLIOTT,D.  472875  C
HASTINGS,G.M.  391  D
BRAMLEY,O.H.  439248  C
*/

Figure 8-24. Updating a REGIONAL(1) data set

Chapter 8: Defining Data Sets for Record Files 123
CRR2: PROC OPTIONS (MAIN);

DCL STOCK FILE RECORD KEYED ENV(REGIONAL(2)),
   LOANS FILE RECORD KEYED ENV(REGIONAL(2)),
   NUMBER CHAR(4),
   1 BOOK,
   2 AUTHOR CHAR(25),
   2 TITLE CHAR(50),
   2 QTY FIXED DEC(3),
   INTER FIXED DEC(5),
   REGION CHAR(8);

OPEN FILE(LOANS) DIRECT OUTPUT;
CLOSE FILE(LOANS);

ON ENDFILE(SYSIN) GO TO FINISH;

OPEN FILE(SYSIN) DIRECT OUTPUT;

NEXT: GET FILE(SYSIN) LIST(NUMBER,BOOK);
   INTER=(NUMBER-1000)/9; /* INTERMEDIATE FIXED DECIMAL ITEM */
   REGION=INTER; /* USED TO ENSURE CORRECT PRECISION */
   WRITE FILE(STOCK) FROM(BOOK) KEYFROM(NUMBER||REGION);
   GO TO NEXT;

FINISH: CLOSE FILE(STOCK);
END CRR2;

/*
GO.LOANS DD DSN=LOANS2,UNIT=2311,SPACE=(19,100),DCB=(RECFM=F,
   BLSIZE=19,DSORG=DA,KEYLEN=7),DISP=(NEW,CATLG),
   VOLUME=SER=D186
GO.STOCK DD DSN=STOCK2,UNIT=2311,SPACE=(19,100),DCB=(RECFM=F,
   BLSIZE=19,DSORG=DA,KEYLEN=7),DISP=(NEW,CATLG),
   VOLUME=SER=D186
GO.SYSIN DD *
'1015' 'W.SHAKEPEARE' 'MUCH ADO ABOUT NOTHING' 1
'1214' 'L.CARROLL' 'THE HUNTING OF THE SNARK' 1
'3079' 'G.FLAUBERT' 'MADAME BOVARY' 1
'3083' 'V.M.HUGO' 'LES MISERABLES' 2
'3085' 'J.K.JEROME' 'THREE MEN IN A BOAT' 2
'4295' 'W.LANGLAND' 'THE BOOK CONCERNING PIERS THE PLOWMAN' 1
'5999' 'O.KHAYYAM' 'THE RUBAIYAT OF OMAR KHAYYAM' 3
'6591' 'F.RABELAIS' 'THE HEROIC DEEDS OF GARGANTUA AND PANTAGRUEL' 1
'8362' 'H.D.THOREAU' 'WALDEN, OR LIFE IN THE WOODS' 1
'9765' 'H.G.WELLS' 'THE TIME MACHINE' 3
*/

Figure 8-25. Creating a REGIONAL(2) data set

REGIONAL(2) figure is in the additional processing required for the analysis of the records read from the data set STOCK3. The records are read into a varying-length character string and a based structure is overlaid on the string so that the data in the record can be extracted.

TELEPROCESSING

Teleprocessing in PL/I is provided by an extension of record-oriented transmission with the addition of the TRANSIENT file attribute and of the PENDING condition. The compiler provides a link between PL/I message processing programs (MPPs) and the Telecommunications Access Method (TCAM) of the operating system.

A TCAM message control program (MCP) handles messages originating from and destined for a number of remote terminals, each of which is identified by a terminal name carried with the message. These messages are transmitted to and from your PL/I message processing program by means of queues in main storage. (These queues are
DUR2: PROC OPTIONS(MAIN);
DCL 1 RECORD,
   2 (ISSUE, REMINDER) CHAR(6),
   SYSIN FILE RECORD INPUT SEQUENTIAL,
   LOANS FILE RECORD UPDATE DIRECT KEYED ENV(REGIONAL(2)),
   CARD CHAR(80),
   DATE BUILTIN,
   BOOK CHAR(4) DEF CARD POS(1),
   READER CHAR(3) DEF CARD POS(10),
   CODE CHAR(1) DEF CARD POS(20),
   REGION CHAR(8) DEF CARD POS(5);

ON ENDFILE(SYSIN) GO TO FINISH;

OPEN FILE(SYSIN), FILE(LOANS);
ISSUE, REMINDER = DATE;

NEXT:
READ FILE(SYSIN) INTO(CARD);
IF CODE='I' THEN WRITE FILE(LOANS) FROM(RECORD)
   KEY FROM READER| BOOK | REGION;
ELSE IF CODE='R' THEN DELETE FILE(LOANS)
   KEY READER | BOOK | REGION;
ELSE IF CODE='A' THEN REWRITE FILE(LOANS) FROM(RECORD)
   KEY READER | BOOK | REGION;
ELSE PUT FILE(SYSPRINT) SKIP LIST
   ('INVALID CODE:', BOOK, READER);
GO TO NEXT;

FINISH: CLOSE FILE(SYSIN), FILE(LOANS);
END DUR2;

MESSAGE PROCESSING PROGRAM (MPP)

This program receives the terminal message as input and produces output according to the data in the message. You can code this program in PL/I.

An MPP is not mandatory at teleprocessing installations, as for example, an MCP is. If the messages you transmit do not require processing, because they are only switched between terminals, an MPP is not required. However, you can pass data to a problem program and you can receive the output with a minimum of delay, and most installations are likely to have a set of processing programs available for these purposes. These programs are stored as load modules, either in main storage or in a library in auxiliary storage.

Figure 8-26. Updating a REGIONAL(2) data set directly supported by corresponding queues on a direct-access device in auxiliary storage. Your PL/I program has access only to the main storage queues by means of a single buffer for each file.)

The exact message format (specified to the compiler by means of the ENVIRONMENT attribute) depends on the MPP. A message may be a complete unit, or may consist of a number of records so that it can be split up for processing. You must have this message format information to enable you to write the message processing program. Full information on how to write an MPP is given in the language reference manual for this compiler. A full account of TCAM procedure is given in the OS: TCAM Message Processing Program Services and OS: TCAM Message Control Program publications.
//OPT8#27 JOB
//STEP1 EXEC PLIXCLG
//PLI.SYSIN DD *
SUR2: PROC OPTIONS (MAIN);

DCL LOANS FILE RECORD SEQUENTIAL UPDATE KEYED ENV(REGIONAL(2)),
STOCK FILE RECORD DIRECT INPUT KEYED ENV(REGIONAL(2)),
(TODAY, LASMTH) CHAR(6),
YEAR PIC '99' DEF LASMTH,
MONTH PIC '99' DEF LASMTH POS(3),
1 RECORD,
2 (ISSUE, REMINDER) CHAR(6),
DATE BUILTIN,
LOANKEY CHAR(7),
READER CHAR(3) DEF LOANKEY POS(1),
BKNO CHAR(4) DEF LOANKEY POS(4),
INTER FIXED DEC(5),
REGION CHAR(8),
1 BOOK,
2 AUTHOR CHAR(25),
2 TITLE CHAR(50),
2 QTY FIXED DEC(3);

TODAY, LASMTH = DATE;
IF MONTH = '01' THEN DO;
    MONTH = '12';
    YEAR = YEAR - 1;
END;
ELSE MONTH = MONTH - 1;
OPEN FILE (LOANS), FILE (STOCK);

ON ENDFILE(LOANS) GO TO FINISH;

NEXT: READ FILE (LOANS) INTO (RECORD) KEYTO (LOANKEY);
IF REMINDER < LASMTH THEN DO;
    REMINDER = TODAY;
    REWRITE FILE (LOANS) FROM (RECORD);
    INTER = (BKNO - 1000) / 9; /* INTERMEDIATE FIXED DECIMAL ITEM */
    REGION = INTER; /* USED TO ENSURE CORRECT PRECISION */
    READ FILE (STOCK) INTO (BOOK) KEY (BKNO || REGION);
    PUT FILE (SYSPRINT) SKIP (4) EDIT (READER, AUTHOR, TITLE)
        (A, SKIP (2));
END;
GO TO NEXT;
FINISH: CLOSE FILE (LOANS), FILE (STOCK);
END SUR2;
/*
//GO.LOANS DD DSN=LOADS2, DISP=OLD
//GO.STOCK DD DSN=STOCK2, DISP=OLD
Figure 8-27. Updating a REGIONAL(2) data set sequentially

HOW TO RUN AN MPP

An example of an MPP and the job control language required to create it is shown in
Figure 8-31. The EXEC statement invokes
the cataloged procedure PLIXCL to compile
and link edit the PL/I message processing
program. The appropriate TCAM modules are
included in the program by the linkage
editor. The load module produced is stored
in the partitioned data set SYS1.MSGLIB
under the member name MPPROC.

MPP is declared as a teleprocessing file
that can process messages up to 100 bytes
long. Similarly OUTMSG is declared as a
teleprocessing file that can receive
messages up to 500 bytes long.

The READ statement gets a record (that
is, a message) from the queue. The
terminal identifier is inserted into the
KEYTO character string. The record is
placed in the INDATA variable for
processing. The appropriate READ SET
statement could also have been used here.
DCL STOCK FILE RECORD KEYED ENV(REGIONAL(3)),
1 CARD,
  2 NUMBER CHAR(4),
  2 AUTHOR CHAR(25) VAR,
  2 TITLE CHAR(50) VAR,
  2 QTY1 FIXED DEC(3),
(L1,L2,X) FIXED DEC(3),
  1 BOOK CTL,
  2 (L3,L4) FIXED DEC(3),
  2 QTY2 FIXED DEC(3),
  2 DESCN CHAR(X) VAR,
INTER FIXED DEC(5),
REGION CHAR(8);

ON ENDFILE(SYSIN) GO TO FINISH;

OPEN FILE(STOCK) SEQUENTIAL OUTPUT;

NEXT: GET FILE(SYSIN) LIST(CARD);
L1=LENGTH(AUTHOR);
L2=LENGTH(TITLE);
X=L1+L2;
ALLOCATE BOOK;
L3=L1;
L4=L2;
QTY2=QTY1;
DESCN=AUTHOR/TITLE;
INTER=(NUMBER-1000)/225; /* INTERMEDIATE FIXED DECIMAL */
REGION=INTER; /* ITEM USED TO ENSURE CORRECT PRECISION */
WRITE FILE(STOCK) FROM (BOOK) KEY FROM (NUMBER)/REGION);
FREE BOOK;
GO TO NEXT;
FINISH: CLOSE FILE(STOCK);
END CRR3;

/*
//GO.STOCK DD DSN=STOCK3,UNIT=2311,SPACE=(TRK,(40,5)),DCB=(RECFM=U,
  BLKSIZE=110,DSORG=DA,KEYLEN=4),DISP=(,CATLG),VOL=SER=D186
//GO.SYSIN DD *
'1015' 'W.SHAKESPEARE' 'MUCH ADO ABOUT NOTHING' 1
'1214' 'L.CARROLL' 'THE HUNTING OF THE SNARK' 1
'3079' 'G.FLAUBERT' 'MADAME BOVARY' 1
'3083' 'V.M.HUGO' 'LES MISERABLES' 2
'3085' 'J.K.JEROME' 'THREE MEN IN A BOAT' 2
'4295' 'W.LANGLAND' 'THE BOOK CONCERNING PIERS THE PLOWMAN' 1
'5999' 'O.KHAYYAM' 'THE RUBAIYAT OF OMAR KHAYYAM' 3
'6591' 'F.RABELAIS' 'THE HEROIC DEEDS OF GARGANTUA AND PANTAGRUEL' 1
'8362' 'H.D.THOREAU' 'WALDEN, OR LIFE IN THE WOODS' 1
'9765' 'H.G.WELLS' 'THE TIME MACHINE' 3
/*

Figure 8-28. Creating a REGIONAL(3) data set
/*

Figure 8-29. Updating a REGIONAL(3) data set directly
*/
Figure 8-30. Updating a REGIONAL(3) data set sequentially
// JOB
// EXEC PLIXCL
//PLI.SYSIN DD *
|
| MPPROC: PROC OPTIONS(MAIN);
| DCL MPP FILE RECORD KEYED TRANSIENT ENV(TP(M)RECSIZE(100)),
| OUTMSG FILE RECORD KEYED TRANSIENT ENV(TP(M)RECSIZE(500)),
| INDATA CHAR(100),
| OUTDATA CHAR(500),
| TKEY CHAR(6);
|
| OPEN FILE(MPP) INPUT,FILE(OUTMSG) OUTPUT;
|
| READ FILE(MPP) KEYTO(TKEY) INTO(INDATA);
|
| WRITE FILE(OUTMSG) KEYFROM(TKEY) FROM(OUTDATA);
|
| ENDP: CLOSE FILE(MPP),FILE(OUTMSG);
| END MPPROC;
|
|//*
|LKED.SYSLMOD DD DSNAMESYS1.MSGLIB(MPPROC),...
|
|Figure 8-31. PL/I message processing program
|
|The WRITE statement puts the data in OUTDATA into the destination queue; the terminal identifier is taken from the character string in TKEY. An appropriate LOCATE statement could also have been used.
|
|Once the load module has been stored on a direct-access device it can be restored for execution at any time. The job control statements to perform this might be:
|
// JOB
//JOBLIB DD DSNAMESYS1.MSGLIB,DISP=SHR
// EXEC PGM=MPPROC
|//MPP DD QNAME=...
|//OUTMSG DD QNAME=...
//SYSPRINT DD SYSOUT=A
|
|The JOBLIB DD statement is required to make SYS1.MSGLIB available so that the operating system can find the program MPPROC. The DD statement with the name DD1 associates the PL/I file with the main storage queue name (MPP).
Chapter 9: Virtual Storage Access Method (VSAM)

The Virtual Storage Access Method (VSAM) is both an access method and a form of data set organization. VSAM is available only to System/370 users. A virtual storage (or relocate) version of the operating system is required.

VSAM data sets can reside only on the following direct access storage devices:

- IBM 2305
- IBM 2314
- IBM 2319
- IBM 3330
- IBM 3340

VSAM Data Sets

There are two types of VSAM data sets: entry sequenced data sets (ESDS) and key sequenced data sets (KSDS). Key sequenced data sets have an associated index; a KSDS together with its index is referred to as an indexed VSAM data set. An entry sequenced data set has no associated index.

All VSAM data sets are cataloged, either in a master catalog or in a user catalog. The catalog entry is made when the data set is "defined", and remains until the data set is "deleted".

DATA FORMAT

The unit of data that is transmitted between a PL/I program and a VSAM data set is called a logical record. Logical records have no defined record format; VSAM will accept records of any length up to a maximum value that is specified when the data set is defined.

Logical records are grouped together in control intervals, and control intervals in turn are grouped in control areas. The sizes of control intervals and control areas are selected by the system to make optimum use of the particular storage device that is being used.

KEY SEQUENCED DATA SETS

Figure 9-1 illustrates the structure of a key sequenced data set. Each control interval contains system control information, one or more logical records, and some free space to allow for the later addition of records. The amount of free space in a control interval can be specified as a percentage of the total space when the data set is defined (see "Creating VSAM Data Sets" later in this chapter). Similarly, the user can specify how many empty control intervals are to be left in each control area to allow for future additions.

Logical records in a KSDS are ordered in the collating sequence of an embedded key. The order is maintained when records are inserted into an existing data set, and when existing records are increased or decreased in length. Deleted records are physically deleted from the data set. The free space specification facility improves the performance of update operations by minimizing the moving of records and the splitting of control intervals and areas when records are added or increased in length.

Associated with each KSDS is an index data set. The index is in the form of a tree structure that gives rapid access to a specified key value. The lowest level of the index data set is known as the "sequence set", and the remaining levels are known as the "index set". For direct access, a logical record with a particular key is located by means of a search through successive levels of the index data set. This process is illustrated in Figure 9-2. For sequential access, only the sequence set is used.

ENTRY SEQUENCED DATA SETS

The structure of an entry sequenced data set is similar to that of a key sequenced data set in that it contains logical records within control intervals within control areas. However, the logical records are stored in the order in which they are submitted when the data set is created, and records cannot be subsequently added (except at the end of the data set), deleted, or changed in length. Any unused space which exists in a control interval is thus wasted space.
Figure 9-1. Structure of Key Sequenced Data Set

LR = logical record
FS = free space
SC = system control information

Figure 9-2. Indexed VSAM Data Set
Operations on VSAM Data Sets

ACCESS METHOD SERVICES

Access Method services is a utility program that enables various operations to be performed on VSAM data sets. It is used to define VSAM data sets, to delete them, to print out their contents, and so on. A full description of the use and syntax of Access Method services is given in OS/VS Access Method Services, Order No. GC35-0009. The principal functions are listed in Figure 9-3.

DEFINE To create VSAM catalogs and data set entries within the VSAM catalogs.
ALTER To change VSAM catalog entries.
DELETE To remove entries from the VSAM catalog.
LISTCAT To list entries within VSAM catalogs.
REPRO To copy the contents of data sets into other data sets.
PRINT To print the contents of data sets.
EXPORT To produce backup or portable copies of VSAM data sets.
IMPORT To accept backup or portable copies of VSAM data sets.
VERIFY To check the end-of-file information in a catalog for correspondence to the physical end of the data set.

Figure 9-3. The principal Access Method Services functions

CREATING VSAM DATA SETS

Before a VSAM data set can be created, VSAM data space must be available on a suitable direct access storage device. Space can be obtained by using the DEFINE command of Access Method Services; this space is then owned by VSAM and is used as a space pool for the creation of VSAM data sets.

When VSAM data space is available, the characteristics of the VSAM data set that is to be created are defined by means of a further DEFINE command. This causes an entry for the VSAM data set to be made in the VSAM master catalog, and the space requested for the data set to be obtained from the available VSAM space and formatted. The data set can then be "loaded" with data by the application program.

The use of the DEFINE command for these purposes is illustrated in the examples later in this chapter.

Note: The DEFINE command for the data set may specify either SUBALLOCATION or UNIQUE. SUBALLOCATION, which is the standard default, specifies that the space for the data set is to be a suballocation of VSAM space on a specified volume. If UNIQUE is specified, the allocation is made directly on a specified volume, and it is not necessary to obtain VSAM space as described in the preceding paragraphs. In the rest of this chapter it is assumed that SUBALLOCATION is in effect.

Creating a Key Sequenced Data Set

The example in Figure 9-4 illustrates the creation of a key sequenced data set. This example is similar to that shown in Chapter 8 for the creation of an INDEXED data set; the only changes in the PL/I program are the replacement of INDEXED by VSAM in the ENVIRONMENT option of the output file declaration, and a modification to the output record structure so that the records contain embedded keys.

The first job step invokes Access Method Services (PGM=IDCAMS) to obtain space for the data set and to enter its characteristics in the VSAM catalog. The first DEFINE command obtains VSAM data space on volume HUR137. Note that the FILE parameter in this DEFINE statement refers to the DD statement with the ddname DO1. This DEFINE command and the corresponding DD statement may be omitted if sufficient VSAM space is already available on the required volume.

The second DEFINE statement defines the VSAM "cluster" that contains the data set and its associated index data set. It specifies that the data set and its index are to be placed on volume HUR137 and that the name of the data set (the dsname) is TELNO. The definition of the data set includes the information that the embedded key is 20 characters long and is at the front of the logical records, that both the average and the maximum lengths of logical records are 23 bytes, that 20% of the space in each control interval is to be left
Figure 9-4. Creating and Initializing a Key Sequenced Data Set
DCL DIREC FILE RECORD KEYED ENV(WSAM),
   ONCODE BUILTIN,
   OUTREC CHAR(23),
   NUMBER CHAR(3) DEF OUTREC POS(21),
   NAME CHAR(20) DEF OUTREC,
   CODE CHAR(2);
ON ENDFILE(SYIN) 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(SYIN) 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;
/*
//GO.DIREC DD DSN=TELNO,DISP=OLD
//GO.SYSIN DD *
NEWMAN,M.W. 516
GOODFELLOW,D.T. 889
MILES,R.
HARVEY,C.D.W. 209
BARTLETT,S.G. 183
CORY,G.
READ,K.M. 001
PITT,W.H.
ROLF,D.F.
ELLIOTT,D. 291
HASTINGS,G.M.
BRAMLEY,O.H. 439
*/

Figure 9-5. Updating a Key Sequenced Data Set
Figure 9-6. Creating an Entry Sequenced Data Set

free, and that 30% of the control intervals in each control area are also to be left free.

The following job steps compile, link-edit, and execute the PL/I program that loads the initial information into the data set.

Note that the only information that need be supplied in the DD statement for the data set is the data set name that was defined in the DEFINE statement, together with the data set disposition (DISP=).

Accessing a Key Sequenced Data Set

The program in Figure 9-5 updates the data set that was created in Figure 9-4 and prints out its new contents. This example is similar to that shown in Chapter 8 for updating an INDEXED data set.
Creating and Accessing an Entry Sequenced Data Set

The example in Figure 9-6 illustrates the creation of an entry sequenced data set. The example is similar to that given in Chapter 8 for the creation of a CONSECUTIVE data set.

The first job step in the example invokes Access Method Services to define and catalog the data set. It is assumed in this example that VSAM already "owns" data space on the specified volume. Note that the VSAM "cluster" in this case contains only the ESDS. NONINDEXED must be specified in the DEFINE statement, to indicate that the records contain no keys and that an ESDS is required.

The PL/I program in this example merges the contents of two existing data sets, DS1 and DS2, and writes them onto the VSAM data set defined in the previous job step. Each of the original data sets contains 8-byte fixed length records arranged in EBCDIC collating sequence.

DD Statements for VSAM Data Sets

This section describes the minimum information that must appear in DD statements for VSAM data sets. The additional facilities that are available through the job control language are also listed; further information on these facilities is given in OS/VS VSAM Programmer’s Guide, Order No. GC26-306.

If a DEFINE command is used to obtain VSAM data space on a particular volume, a DD statement must be provided to define the volume. The ddname of this DD statement must appear in the FILE parameter of the DEFINE command. The DD statement must contain the UNIT, VOLUME, and DISP parameters. For example, if the DEFINE command is:

```
DEFINE SPACE(.... FILE(DDN).....)
```

the corresponding DD statement must have the form:

```
//DDN DD UNIT=xxxx, VOL=SER=yyyy, DISP=OLD
```

Once a data set has been defined using the DEFINE command, it can be accessed by specifying the dsname and a disposition of OLD. Additional parameters can, however, be specified. For VSAM data sets, the DCB parameter is invalid; its place is taken by the AMP parameter. The AMP parameter has the following format:

```
AMP='subparameter,subparameter,...'
```

The valid subparameters of the AMP parameter are:

- **BUFS=number** specifies the amount of storage to be set aside for VSAM buffers.
- **BUFND=number** specifies the number of data buffers required.
- **BUFNI=number** specifies the number of index buffers required.
- **RECFM=format** specifies record format. The use of this subparameter is described under the heading "The Compatibility Interface" later in this chapter.

- **STRNO=number** specifies how many concurrent data-set positioning requests VSAM must be prepared to handle. For convenience, the PL/I programmer can make STRNO equal to the number of PL/I files that will be open on the data set at the same time.

The Compatibility Interface

PL/I programs that are written to create and access ISAM data sets (ENV(INDEXED)) can be used to access VSAM key sequenced data sets without alteration. Two types of access are possible: "native" access, in which the data set is accessed exactly as though the file had been declared with ENV(VSAM), and access via the ISAM/VSAM compatibility interface.

The fact that the data set being
accessed is a VSAM data set is detected by the PL/I library routines when the file is opened, and the required support is provided. This support will normally be native VSAM support; in order to force use of the compatibility interface, the user must code either RECFCM=F|FB|VB|V or OPTCD=L in the AMP parameter of the DD statement for the data set. Use of the compatibility interface must be forced if any of the following situations exists:

- The PL/I program uses records with non-embedded keys.
- The user requires the lengths of records being read or written to be checked against the record length specified for the data set; the RECORD condition is to be raised if an incorrect-length record is encountered.
- Deleted records are to be retained in the data set (ISAM deletion). If deleted records are to be deleted from the data set (VSAM deletion), but the compatibility interface must be used for some other reason, the programmer must specify OPTCD=IL.
- The number of channel programs specified (NCP) is greater than one. Note, however, that the compatibility interface is obtained automatically in this case if a file declared with ENV(INDEXED) is opened on a VSAM KSDS.

If the compatibility interface is used, and RECFCM is not specified either in the program or in the AMP parameter, the default is RECFCM=V.

If a PL/I file declared with ENV(INDEXED) is used to load a VSAM KSDS with F-format records, and the key is not embedded, the record size specified on the DEFINE command for the VSAM data set must be equal to the length of the user record plus the length of the key.

Password Protection of VSAM Data Sets

VSAM provides a password protection option that enables VSAM data sets to be protected against unauthorized access or deletion. Passwords are specified as a parameter of the DEFINE command when a VSAM data set is defined. In order to gain access to a password-protected data set, the PL/I programmer must specify the correct password in the the ENVIRONMENT attribute of the PL/I file.

There are three levels of password protection of interest to the PL/I programmer:

1. Master Password - Specifying this password allows the user to perform any operation on a data set or on the index and catalog records associated with it.
2. Update Password - Specifying this password allows the user to retrieve, update, delete, or insert records in a data set.
3. Read-Only Password - Specifying this password allows the user to retrieve records from a data set, but not to update, delete, or insert records.

Note that password protection is effective only if the catalog that contains the data set is itself password protected.

Sharing VSAM Data Sets

SHARING BETWEEN JOBS

Independent jobs running in the same system may share VSAM data sets provided that both jobs specify DISP=SHR in their DD statements for the data set. The type of sharing that is to be allowed on any particular data set can be specified in the DEFINE command when the data set is defined. The following four types of sharing are possible:

1. The data set may be used by one job for output or by any number of jobs for input.
2. The data set may be used by one job for output and by any number of jobs for input.
3. The data set may be fully shared, the user being completely responsible for read and write integrity.
4. As (3) above, but VSAM will refresh the buffers for each request.

SHARING BETWEEN SUBTASKS IN A JOB

Subtasks can share VSAM data sets either through separate DD statements or through the same DD statement.

If separate DD statements are used, the rules are the same as those for sharing between jobs.
If a single DD statement is used, the data set can be fully shared. The value of STRNO specified in the AMP parameter of the DD statement should be equal to the number of files that will be open on the data set concurrently.
Within the IBM operating system, the terms "partitioned data set" and "library" are used synonymously to signify a type of data set that can be used for the storage of other data sets (usually programs in the form of source, object or load modules). A library must be stored on direct-access storage and be wholly contained in one volume. It contains independent, consecutively-organized, data sets, called members. Each member has a unique name, not more than eight characters long, which is stored in a directory that is part of the library. All the members of one library must have the same data characteristics because only one data set label is maintained.

Members can be created individually until there is insufficient space left for a new entry in the directory, or until there is insufficient space for the member itself. Members can be accessed individually by specifying the member name. DD statements or ALLOCATE commands are used to create and access members.

Members can be deleted by means of the IBM utility program IEHPROGM. This deletes the member name from the directory so that the member can no longer be accessed; but the space occupied by the member itself cannot be used again unless the library is recreated using, for example, the IBM utility program IEBCOPY. An attempt to delete a member by using the DISP parameter of a DD statement will cause the whole data set to be deleted.

Types of Library

The following types of library may be used with a PL/I program:

- The system program library SYS1.LINKLIB. This can contain all system processing programs such as compilers and the linkage editor.

- Private program libraries. These usually contain user-written programs. It is often convenient to create a temporary private library to store the load module output from the linkage editor until it is executed by a later job step in the same job. The library will be deleted at the end of the job. Private libraries are also used for automatic library call by the linkage editor and the loader.

- The system procedure library, SYS1.PROCLIB. This contains the job control procedures that have been cataloged for your installation.

How to Use a Library

The ways in which the libraries described above can be used are described in the following sections.

BY THE LINKAGE EDITOR OR LOADER

The output from the linkage editor is usually placed on a private program library.

The automatic call library used as input to the linkage editor or loader (see Chapter 5) can be SYS1.LINKLIB, a private program library, or a subroutine library.

In each case, the processing of directory entries is performed by the operating system.

When you are adding a member to a library, you must specify the member name as follows:

- When a single module is produced as output from the linkage editor, the member name can be specified as part of the data set name (see later in this chapter).

- When more than one module is produced as output from the linkage editor, the member name for the first module can be specified as part of the data set name or in the NAME option or NAME control statement. The member names for the subsequent modules must be specified in the NAME option or the NAME control statement.

BY THE OPERATING SYSTEM

When you request the execution of a load module in an EXEC statement or CALL
command, the operating system must be able to retrieve the load module from a library. For a CALL command, this library is specified explicitly or implicitly in the command. For an EXEC statement, the following rules apply.

The operating system will assume the load module is a member of SYS1.LINKLIB, and will search in the directory for that library for the name you have specified, unless you have also specified that the load module is in a private library, in one of the following ways.

If the load module has been added to the private library in a previous step of the job (usually a link-edit step) and the member name was specified as part of the data set name, then you can refer, in the EXEC statement, to the DD statement defining the library instead of specifying the load module name. The library must have been given the disposition P,ASS.

If the load module exists on the private library before the job starts, then you have several ways of defining the library. You can define the library in a DD statement, with the ddname JOBLIB, immediately after the JOB statement. This library will be used in place of SYS1.LINKLIB for all the steps of the job. If any load module is not found on the private library, the system will then look for it on SYS1.LINKLIB.

You can define the library in a DD statement with the ddname STEPLIB, at any point in the job control procedure. The private library will be used in place of SYS1.LINKLIB, or any library specified in a JOBLIB DD statement, for the job step in which it appears (though it can also be "passed" to subsequent job steps in the normal way). If any load module is not found on the private library, the system will look for it on the library specified in the JOBLIB DD statement (if used) or on SYS1.LINKLIB. The STEPLIB DD statement can be used in a cataloged procedure.

Alternatively, if you specify SYS1.LINKLIB in the JOBLIB or STEPLIB DD statements, and then concatenate the private library to it, the private library will be used only if a load module cannot be first found on SYS1.LINKLIB.

BY YOUR PROGRAM

Libraries can be used directly by a PL/I program.

If you are adding a new member to a library, its directory entry will be made by the operating system when the associated file is closed, using the member name specified as part of the data set name.

If you are accessing a member of a library, its directory entry can be found by the operating system from the member name that you specify as part of the data set name.

More than one member of the same library can be processed by the same PL/I program, but only one such output file can be open at any one time. Different members are accessed by giving the member name in a DD statement.

Creating a Library

To create a library include in your job step a DD statement containing the information given in Figure 10-1. The information required is similar to that for a consecutively-organized data set (see Chapter 8) except for the SPACE parameter.

```
Figure 10-1. Information required when creating a library

<table>
<thead>
<tr>
<th>Information Required</th>
<th>Parameter of DD statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type of device that will process the library</td>
<td>UNIT=</td>
</tr>
<tr>
<td>Serial number of the volume that will contain the library</td>
<td>VOLUME=SER</td>
</tr>
<tr>
<td>Name of the library</td>
<td>DSNNAME=</td>
</tr>
<tr>
<td>Amount of space required for the library</td>
<td>SPACE=</td>
</tr>
<tr>
<td>Disposition of the library</td>
<td>DISP=</td>
</tr>
</tbody>
</table>
```

SPACE PARAMETER

The SPACE parameter in a DD statement that defines a library must always be of the form:

```
SPACE=(units,(quantity, increment, directory))
```

Although you can omit the third term (increment), indicating its absence by a comma, the last term, specifying the number
Creating a Library Member

The amount of auxiliary storage required for a library depends on the number and sizes of the members to be stored in it and on how often members will be added or replaced. (Space occupied by deleted members is not released.) The number of directory blocks required depends on the number of members and the number of aliases. Although you can specify an incremental quantity that will allow the operating system to obtain more space for the data set if necessary, both at the time of creation and when new members are added, the number of directory blocks is fixed at the time of creation and cannot be increased.

If the data set is likely to be large or you expect to do a lot of updating, it might be best to allocate a full volume. Otherwise, make your estimate as accurate as possible to avoid wasting space or time recreating the data set.

The number of directory entries that a 256-byte directory block can contain depends on the amount of user data included in the entries. The maximum length of an entry is 74 bytes, but the entries produced by the linkage editor vary in length between 34 bytes and 52 bytes, which is equivalent to between four and seven entries per block.

For example, the DD statement:

```
//PDS DD UNIT=2311,VOLUME=SER=3412,
// DSNAME=ALIB,
// SPACE=(CYL,(50,,10)),
// DISP=(,CATLG)
```

requests the job scheduler to allocate 50 cylinders of the 2311 disk pack with serial number 3412 for a new partitioned data set named ALIB, and to enter this name in the system catalog. The last term of the SPACE parameter requests that part of the space allocated to the data set be reserved for ten directory blocks.

Creating a Library Member

The members of a library must have identical characteristics otherwise you may subsequently have difficulty retrieving them. This is necessary because the volume table of contents (VTOC) will contain only one data set control block (DSCB) for the library and not one for each member. When using a PL/I program to create a member, the operating system creates the directory entry; you cannot place information in the user data field.

When creating a library and a member at the same time, the DD statement must include all the parameters listed under the heading "Creating a Library," earlier in this chapter (although you can omit the DISP parameter if the data set is to be temporary). The DSNAME parameter must include the member name in parentheses. For example, DSNAME=ALIB(MEM1) names the member MEM1 in the data set ALIB. If the member is placed in the library by the linkage editor, you can use the linkage editor NAME statement or the NAME compiler option instead of including the member name in the DSNAME parameter. You must also describe the characteristics of the member (record format, etc.) either in the DCB parameter or in your PL/I program; these characteristics will also apply to other members added to the data set.

When creating a member to be added to an existing library, you will not need the SPACE parameter; the original space allocation applies to the whole of the library and not to an individual member. Furthermore, you will not need to describe the characteristics of the member, since these are already recorded in the DSCB for the library.

To add two or more members to a library in one job step, you must include a DD statement for each member, and you must close one file that refers to the library before you open another.

Examples

The use of the cataloged procedure PLIXC to compile a simple PL/I program and place the object module in a new library named EXLIB is shown in Figure 10-2. The DD statement that defines the new library and names the object module overrides the DD statement SYSLIN in the cataloged procedure. (The PL/I program is a function procedure that, given two values in the form of the character string produced by the TIME built-in function, returns the difference in milliseconds.)

The use of the cataloged procedure PLIXCL to compile and link edit a PL/I program and place the load module in the existing library "PUPPGM" is shown in Figure 10-3. (The PL/I program lists the names of the members of a library.)

To use a PL/I program to add or delete one or more records within a member of a library, you must rewrite the entire member in another part of the library; this is

Chapter 10: Libraries of Data Sets 143
DCL (TIME1,TIME2) CHAR(9),
   H1 PIC '99' DEF TIME1,
   M1 PIC '99' DEF TIME1 POS(3),
   MS1 PIC '99999' DEF TIME1 POS(5),
   H2 PIC '99' DEF TIME2,
   M2 PIC '99' DEF TIME2 POS(3),
   MS2 PIC '99999' DEF TIME2 POS(5),
   ETIME FIXED DEC(7);

IF H2<H1 THEN H2=H2+24:
ETIME=
   (H2*60+M2)*600000+MS2) - ((H1*60+M1)*600000+MS1);
RETURN(ETIME);
END ELAPSE;
/*
Figure 10-2. Creating new libraries for compiled object modules

DCL LINK FILE RECORD SEQUENTIAL INPUT,
   1 DIRBLK,
   2 COUNT BIT(16),
   2 ENTRIES(254) CHAR(1),
   1 ENTRY BASED(A),
   2 NAME CHAR(8),
   1 ENTRY
   1 TTR CHAR(3),
   1 INDIC,
   3 ALIAS BIT(1),
   3 TTRS BIT(2),
   3 USERCT BIT(5),
   (LEN,PTR) FIXED BIN(31);

ON ENDFILE(LINK) GO TO FINISH;

OPEN FILE(LINK);
   NEXTBLK: READ FILE(LINK) INTO(DIRBLK);
   LEN=COUNT;
   PTR=1;
      NEXTENT: A=ADDR(ENTRIES(PTR));
      PUT FILE(SYSPRINT) SKIP LIST(NAME);
      PTR=PTR+12+2*USERCT;
      IF PTR+2>LEN THEN GO TO NEXTBLK;
      GO TO NEXTENT;
   FINISH: CLOSE FILE(LINK);
END MNAME;
/*
Figure 10-3. Placing a load module in an existing library
Figure 10-4. Creating a library member in a PL/I program

Figure 10-5. Updating a library member

rarely an economic proposition, since the space originally occupied by the member cannot be used again. You must use two files in your PL/I program, but both can be associated with the same DD statement. The program shown in Figure 10-5 updates the member created by the program in Figure 10-4; it copies all the records of the original member except those that contain only blanks.

Library Structure

The structure of a library is illustrated in Figure 10-6. The directory of a library is a series of records (entries) at the beginning of the data set; there is at least one directory entry for each member. Each entry contains a member name, the relative address of the member within the library, and a variable amount of user data. The entries are arranged in ascending alphabetic order of member names.

A directory entry can contain up to 62 bytes of user data (information inserted by the program that created the member). An entry that refers to a member (load module)
Note:
Pointers contain relative addresses of locations within member.

Figure 10-6. Structure of a library
written by the linkage editor includes user data in a standard format, described in the manual System Control Blocks.

If you use a PL/I program to create a member, the operating system creates the directory entry for you and you cannot write any user data. However, you can use assembler language macro instructions to create a member and write your own user data; the method is described in the manual Supervisor and Data Management Services.

Directory entries are stored in fixed-length blocks of 256 bytes, each containing a 2-byte count field specifying the number of active bytes in a block, and as many complete entries as will fit into the remaining 254 bytes. The directory is in effect a sequential data set that contains fixed-length unblocked records, and can be read as such.

The program illustrated in Figure 10-3 demonstrates a method of extracting information from directory entries. The program lists the names of all the members of a library; the library must be defined, when the program is executed, in a DD statement with the name LINK.
This chapter describes the standard cataloged procedures supplied by IBM for use with the OS PL/I Optimizing Compiler, explains how to invoke them, and how to make temporary or permanent modifications to them.

A cataloged procedure is a set of job control statements stored in a system library, the procedure library (SYS1.PROCLIB). It comprises one or more EXEC statements, each of which may be followed by one or more DD statements. You can retrieve the statements by naming the cataloged procedure in the PROC parameter of an EXEC statement in the input stream. When the operating system processes this EXEC statement, it replaces it in the input stream with the statements of the cataloged procedure.

The use of cataloged procedures saves time and reduces errors in coding frequently used sets of job control statements. If the statements in a cataloged procedure do not match your requirements exactly, you can easily modify them or add new statements for the duration of a job. It is recommended that each installation review these procedures and modify them to obtain the most efficient use of the facilities available and to allow for installation conventions; refer to "Permanent Modification," later in this chapter.

Invoking a Cataloged Procedure

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

```
//stepname EXEC PROC=PLIXC
```

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

```
//stepname EXEC PLIXC
```

When the operating system meets the name of a cataloged procedure in an EXEC statement, it extracts the statements of the cataloged procedure from the procedure library and substitutes them for the EXEC statement in the input job stream. If you include the parameter MSGLEVEL=1 in your JOB statement, the operating system will include the original EXEC statement in its listing, and will add the statements of the cataloged procedure. In the listing, cataloged procedure statements are identified by XX or X/ as the first two characters; X/ signifies a statement that has been modified for the current invocation of the cataloged procedure.

An EXEC statement identifies a job step, which can require either the execution of a program or the invocation of a cataloged procedure. A cataloged procedure includes one or more EXEC statements, which identify procedure steps. However, an EXEC statement in a cataloged procedure cannot invoke another cataloged procedure; it must request the execution of a program.

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

Multiple Invocation of Cataloged Procedures

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

The linkage editor always places a load module that it creates in the standard data set defined by the DD statement with the name SYSLMOD. In the absence of a linkage editor NAME statement for the NAME compiler...
option), it uses the member name specified in the DSNAME parameter as the name of the module. In the standard cataloged procedures, the DD statement with the name SYSLMOD always specifies a temporary library named &GOSET, and gives the load module the member name GO.

Consider what will happen if, for example, you use the cataloged procedure PLIXCLG twice in a job to compile, link edit, and execute two PL/I programs, and do not name each of the two load modules that will be created by the linkage editor. The linkage editor will name the first load module GO, as specified in the first DD statement with the name SYSLMOD. It will not be able to use the same name for the second load module since the first load module still exists in the library &GOSET; it will allocate a temporary name to the second load module (a name that is not available to your program). Step GO of the cataloged procedure requests the operating system to initiate execution of the load module named in the first DD statement with the name SYSLMOD. Consequently, the first load module will be executed twice and the second not at all.

To prevent this, use one of the following methods:

- Delete the library &GOSET at the end of the step GO of the first invocation of the cataloged procedure by adding a DD statement of the form:

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

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

  //LKEO.SYSLMOD DD DSN=&GOSET(GO1)

  and so on.

- Use the NAME compiler option to give a different name to each load module and change your job control statements to specify the execution of the load modules with these names.

### Dedicated Data Sets

Many of the processing programs in the operating system, including the optimizing compiler and the linkage editor, use temporary workfiles. To avoid allocating data sets for these workfiles each time they are required, an installation using the MVT operating system can dedicate one or more data sets for temporary workfiles, and these remain permanently allocated.

The standard cataloged procedures allow you to assign dedicated data sets to the optimizing compiler and linkage editor. The DD statements for workfiles have the ddname SYSUT1. In these DD statements, the DSNAME parameter is coded:

```
DSNAME=&ddname
```

where "ddname" is the ddname of that DD statement. Your installation must have assigned these names to the dedicated data sets, otherwise you must override the DD statement in the cataloged procedure in order to specify the names used by your installation.

If the system cannot assign the dedicated data set to your job step, it creates a temporary data set instead. For full details of dedicated data sets see the OS System Programmer's Guide.

### Multitasking Using Cataloged Procedures

When you use a cataloged procedure to link edit a multitasking program, you must ensure that the load module includes the multitasking versions of the PL/I resident library subroutines. To enable you to select the appropriate library, the cataloged procedures that invoke the linkage editor and the loader include a symbolic parameter (SLKLBDSN) in the DSNAME parameter of the DD statement SYSLIB, which defines the data set to be used as the automatic call library. This data set is described in chapter 5. The default value of this symbolic parameter is SYS1.PLIBASE, which is the name of the non-multitasking ("base") library.

To ensure that the multitasking library (SYS1.PLIBASE) is searched before the base library, include the parameter LKLBDSN='SYS1.PLIBASE' in the EXEC statement that invokes the cataloged procedure; for example:

```
//STEPSA EXEC PLIXCLG,LKLBDSN='SYS1.PLIBASE'
```

The DD statement SYSLIB is always followed in the standard cataloged procedures by another, unnamed, DD statement that includes the parameter DSNAME=SYS1.PLIBASE. The effect of this statement is to concatenate the base library with the multitasking library, if the latter is used; the base library can
then be searched for any subroutine common to multitasking and non-multitasking and therefore not included in the multitasking library. When the non-multitasking library is selected, the second DD statement has no effect.

The use of the symbolic parameter $LKLBDSN means that for non-multitasking programs, SYS1.PLIBASE is concatenated with itself. This has no effect other than a very small increase in job scheduling time, but does avoid the need for different cataloged procedures for link editing multitasking and non-multitasking programs.

### Modifying Cataloged Procedures

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

Alternatively, you can modify a cataloged procedure permanently by rewriting the job control statements that are stored in the procedure library. Permanent modification should be made only by system programmers responsible for maintaining the procedure library. Some of the considerations that may influence their decisions to modify the standard cataloged procedures are discussed below.

### TEMPORARY MODIFICATION

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

### EXEC Statement

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

- **PARM** applies to the first procedure step and nullifies any other PARM parameters.
- **COND** and **ACCT** apply to all the procedure steps.
- **TIME** and **REGION** apply to all the procedure steps and override existing values.

For example, the statement:

```
//stepname EXEC PLIXCLG,PARM='SIZE(MAX)', REGION=144K
```

invokes the cataloged procedure PLIXCLG, substitutes the option **SIZE(MAX)** for **OBJECT** and **NODECK** in the EXEC statement for procedure step **PLI**, and nullifies the **PARM** parameter in the EXEC statement for procedure step **LKED**; it also specifies a region size of **144K** for all three procedure steps.

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

```
//stepname EXEC PROC=PLIXCLG,PARM='SIZE(MAX)',REGION.PLI=144K
```

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

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

```
//stepname EXEC PLIXCLG,PARM.LRED=
```

### DD Statement

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

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

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

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

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

Overriding DD statements must follow the EXEC statement that invokes the cataloged procedure in the same order as the corresponding DD statements of the cataloged procedure. DD statements that are being added must follow the overriding DD statements for the procedure step in which they are to appear.

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

**PERMANENT MODIFICATION**

To make permanent modifications to a cataloged procedure, or to add a new cataloged procedure, use the system utility program IEBUPDTE, which is described in the utilities publication. The following paragraphs discuss some of the factors you should have in mind when considering whether to modify the standard cataloged procedures for your installation. For further information on writing installation cataloged procedures see the system programmer's guide.

In general, installation conventions will dictate the options that you include in the FARM, UNIT, and SPACE parameters of the cataloged procedures, and also the blocking factors for output data sets.

If your installation uses the MVT control program of the operating system, you may need to modify some or all of the REGION parameters.

The minimum region size for compilation should be at least 8K bytes larger than the largest value that will be specified in the SIZE compiler option, excluding SIZE(MAX).

In cataloged procedures that invoke the linkage editor, a region size of 106K is specified for the link-edit procedure step.

You can reduce this region size if you are using the 44K F-level linkage editor. In general, the region size should be at least 8K bytes larger than the design size for the particular version of the linkage editor being used. You must alter the region size if you are using the 128K F-level linkage editor.

Under MVT, the operating system requires up to 52K bytes of main storage within a region when initiating or terminating a job step. If you specify a region size of less than 52K bytes, completion of a job may be held up until 52K bytes are available.

The minimum region size used by MVT is dependent on the installation, and is defined at system generation. There is nothing to be gained in reducing the region size below this value.

If your installation uses MFT only, you can delete the REGION parameter from all cataloged procedures, otherwise it will be ignored.

**IBM-Supplied Cataloged Procedures**

The standard PL/I cataloged procedures supplied for use with the optimizing compiler are:

- PLIXC Compile only
- PLIXCL Compile and link edit
- PLIXCLG Compile, link edit, and execute
- PLIXLG Link edit and execute
- PLIXCG Compile, load-and-execute
- PLIXG Load-and-execute
The individual statements of the cataloged procedures are not fully described, since all the parameters are discussed elsewhere in this publication. These cataloged procedures do not include a DO statement for the input data set; you must always provide one. The example shown in Figure 11-1 illustrates the JCL statements you might use to invoke the cataloged procedure PLIXCLG to compile, link edit, and execute, a PL/I program.

```
//COLEGO JOB
//STEPl EXEC PLIXCLG
//PLI.SYSIN DD *

(insert here PL/I program to be compiled)

/*
Figure 11-1. Invoking a cataloged procedure

No IBM-supplied cataloged procedure is provided to produce an object module on punched cards. You can temporarily modify any of the cataloged procedures that have a compile step to produce a punched card output; an example is shown in Figure 11-2.

```

```
//stepname EXEC PLIXCLG,
//PLI.SYSIN DD *
//PLI.SYSIN DD *

/*
Figure 11-2. Modifying a cataloged procedure to produce a punched card output

Compile Only (PLIXC)

This cataloged procedure, shown in Figure 11-3, comprises only one procedure step, in which the options specified for the compilation are OBJECT and NOECK. (IELOOA is the symbolic name of the compiler.) In common with the other cataloged procedures that include a compilation procedure step, PLIXC does not include a DD statement for the input data set; you must always supply an appropriate statement with the qualified

ddname PLI.SYSIN.

The OBJECT option causes the compiler to place the object module, in a form suitable for input to the linkage editor, in the standard data set defined by the DD statement with the name SYSLIN. This statement defines a temporary data set named &LOADSET on a magnetic-tape or direct-access device; if you want to retain the object module after the end of your job, you must substitute a permanent name for &LOADSET (that is, a name that does not commence & & ) and specify KEEP in the appropriate DISP parameter for the last procedure step in which the data set is used.

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

The SPACE parameter allows an initial allocation of 250 eighty-byte records and, if necessary, 15 further allocations of 100 records (a total of 1750 records, which should suffice for most applications).

Compile and Link-edit (PLIXCL)

This cataloged procedure, shown in Figure 11-4, comprises two procedure steps: PLI, which is identical with cataloged procedure PLIXC, and LKED, which invokes the linkage editor (symbolic name IEWL) to link edit the object module produced in the first procedure step.

Input data for the compilation procedure step requires the qualified ddname PLI.SYSIN. The COND parameter in the EXEC statement LKED specifies that this procedure step should be bypassed if the return code produced by the compiler is greater than 9 (that is, if a severe or unrecoverable error occurs during compilation).

The DD statement with the name SYSLIB specifies the PL/I resident library, from which the linkage editor will obtain appropriate modules for inclusion in the load module. The linkage editor always places the load modules it creates in the standard data set defined by the DD statement with the name SYSLMOD. This statement in the cataloged procedure specifies a new temporary library &G0SET, in which the load module will be placed and given the member name GO (unless you specify the NAME compiler option for the

Chapter 11: Cataloged Procedures 153
Figure 11-3. Cataloged procedure PLIXC

```
//PLIXC  PROC
//PLI    EXEC PGM=IELOAA, PARM='OBJECT,NODECK', REGION=100K
//SYSPRINT DD SYSOUT=A, DCB=(RECFM=VBA, LRECL=125, BLKSIZE=629)
//SYSLIN  DD DSN=&&LOADSET, DISP=(MOD,PASS), UNIT=SYSSQ,
//         SPACE=(80,(250,100))
//SYSUT1  DD DSN=&&SYSUT1, UNIT=SYSDA, SPACE=(1024,(60,60),, CONTIG),
//         DCB=BLKSIZE=1024
```

Figure 11-4. Cataloged procedure PLIXCL

```
//PLIXCL  PROC LKLBDSN='SYS1.PLIBASE'
//PLI    EXEC PGM=IELOAA, PARM='OBJECT,NODECK', REGION=100K
//SYSPRINT DD SYSOUT=A, DCB=(RECFM=VBA, LRECL=125, BLKSIZE=629)
//SYSLIN  DD DSN=&&LOADSET, DISP=(MOD,PASS), UNIT=SYSSQ,
//         SPACE=(80,(250,100))
//SYSUT1  DD DSN=&&SYSUT1, UNIT=SYSDA, SPACE=(1024,(60,60),, CONTIG),
//         DCB=BLKSIZE=1024
//LKED    EXEC PGM=IEWL, PARM='XREF,LIST', COND=(9,LT, PLI), REGION=100K
//SYSLIB  DD DSN=LKLBDNSN, DISP=SHR
//        DD DSN=SYS1.PLIBASE, DISP=SHR
//SYSLMOD DD DSN=&&GOSET(GO), DISP=(MOD,PASS), UNIT=SYSDA,
//         SPACE=(1024,(50,20,1),RLSE)
//SYSUT1  DD DSN=&&SYSUT1, UNIT=SYSDA, SPACE=(1024,(200,20)),
//         DCB=BLKSIZE=1024
//SYSPRINT DD SYSOUT=A
//SYSLIN  DD DSN=&&LOADSET, DISP=(OLD,DELETE)
//        DD Ddbname=SYsin
```

Figure 11-5. Cataloged procedure PLIXCLG
Figure 11-6. Cataloged procedure PLIXLG

In specifying a temporary library, the cataloged procedure assumes that you will execute the load module in the same job; if you want to retain the module, you must substitute your own statement for the DD statement with the name SYSLMOD.

The last statement, DDNAME=SYSLIN, illustrates how to concatenate a data set defined by a DD statement with the name SYSLIN with the primary input (SYSLIN) to the linkage editor. You could place linkage editor control statements in the input stream by this means, as described in Chapter 5.

Compile, Link-edit, and Execute (PLIXCLG)

This cataloged procedure, shown in Figure 11-5, comprises three procedure steps, PLI and LKED, which are identical with the two procedure steps of PLIXCL, and GO, in which the load module created in the step LKED is executed. The third procedure step will be executed only if no severe or unrecoverable errors occur in the preceding procedure steps.

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

Link-edit and Execute (PLIXLG)

This cataloged procedure, shown in Figure 11-6, comprises two procedure steps, LKED and GO, which are similar to the procedure steps of the same names in PLIXCLG.

In the procedure step LKED, the DD statement with the name SYSLIN does not define a data set, but merely refers the operating system to the DD statement SYSIN, which you must supply with the qualified dname LKED.SYSIN. This DD statement defines the data set from which the linkage editor will obtain its primary input. Execution of the procedure step GO is conditional on successful execution of the procedure step LKED only.

Compile, Load, and Execute (PLIXCG)

This cataloged procedure, shown in Figure 11-7, achieves the same results as PLIXCLG but uses the loader instead of the linkage editor. However, instead of using three procedure steps (compile, link edit, and execute), it has only two (compile, and load-and-execute). In the second procedure step, the loader program is executed; this program processes the object module produced by the compiler and executes the resultant executable program immediately. Input data for the compilation procedure step requires the qualified dname PLI.SYSIN.

The REGION parameter of the EXEC statement GO specifies 100K bytes. Since the loader requires about 17K bytes of main storage, there are about 83K bytes for your program; if this is likely to be insufficient, you must modify the REGION parameter. The use of the loader imposes certain restrictions on your PL/I program; before using this cataloged procedure, refer to Chapter 5, which explains how to use the loader.

Load and Execute (PLIXG)

This cataloged procedure, shown in Figure 11-8, achieves the same results as PLIXLG.
Cataloged procedure PLIXCG

```apl
//PLIXG PROC LKLBDSN='SYS1.PLIBASE'
//GO EXEC PGM=LOADER,PARM='MAP,PRINT',REGION=100K
//SYSLIB DD DSN=&&LKLBDSN,DISP=SHR
//DD DSN=SYS1.PLIBASE,DISP=SHR
//SYSOUT DD SYSOUT=A
//SYSPRINT DD SYSOUT=A

Figure 11-8. Cataloged procedure PLIXG
```

but uses the loader instead of the linkage editor. However, instead of using two procedure steps (link edit and execute), it has only one. In this procedure step, the loader program is executed. This program processes and executes an object module placed in the data set defined by a DD statement with the name SYSLIN; you must supply this statement with the qualified name GO.SYSLIN.

The REGION parameter of the EXEC statement GO specifies 100K bytes. Since the loader requires about 17K bytes of main storage, there are about 83K bytes for your program; if this is likely to be insufficient, you must modify the REGION parameter. The use of the loader imposes certain restrictions on your PL/I program; before using this cataloged procedure, refer to Chapter 5, which explains how to use the loader.
Program checkout is the application of diagnostic and test processes to a program. You should give adequate attention to program checkout during the development of a program so that:

- A program becomes fully operational after the fewest possible test runs, thereby minimizing the time and cost of program development.
- A program is proved to have fulfilled all the design objectives before it is released for production work.
- A program has complete and clear documentation to enable both operators and program maintenance personnel to use and maintain the program without assistance from the original programmer.

The data used for the checkout of a program should be selected to test all parts of the program. Whilst the data should be sufficiently comprehensive to provide a thorough test of the program, it is easier and more practical to monitor the behaviour of the program if the volume of data is kept to a minimum.

### Conversational Program Checkout

The optimizing compiler can be used in conversational mode when writing and testing programs at a terminal. The conversational features are available to users where the TSO (Time Sharing Option) facilities of the operating system are present. The conversational facilities enable you to enter a PL/I program from a terminal, through which you will receive diagnostic messages for the compilation. You can also communicate with the program during execution using PL/I files associated with the terminal. Thus a PL/I program can be checked out during its construction, thereby saving a substantial amount of elapsed time that can occur between test compilation and execution runs in batched processing.

The PL/I program is entered and processed using the PLI, EDIF, and other commands and features described in the publication *OS TSO: PL/I Optimizing Compiler*.

### Compile-time Checkout

At compile time, both the preprocessor and the compiler can produce diagnostic messages and listings according to the compiler options selected for a particular compilation. The listings and the associated compiler options are discussed in Chapter 4. The diagnostic messages produced by the optimizing compiler are identified by a number prefixed "IEL". These diagnostic messages are available in both a long form and a short form. The long messages are designed to be as self-explanatory as possible. The short messages are designed for reproduction at a terminal when the compiler is being used in a TSO environment. The short messages are obtained by specifying the SMESSAGE compiler option. Each message is reproduced in the publication: *OS PL/I Optimizing Compiler Messages*. This publication includes explanatory notes, examples, and any action to be taken.

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

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

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

The NOSYNTAX option specified with the
severity level "W", "E", or "S" will cause
compilation of the output from the PL/I
preprocessor, if used, to be suppressed
prior to the syntax-checking stage should
the preprocessor issue diagnostic messages
at or above the severity level specified in
the option.

The NOCOMPILE option specified with the
severity level "W", "E", or "S" will cause
compilation to be suppressed after the
syntax-checking stage if syntax checking or
preprocessing causes the compiler to issue
diagnostic messages at or above the
severity level specified in the option.

**Linkage Editor Checkout**

When using the linkage editor, check
particularly that any required overlay
structuring and incorporation of additional
object and load modules have been performed
correctly. Diagnostic messages produced by
the linkage editor are prefixed "IEW".
These messages are fully documented in the
publication: OS Linkage Editor and Loader.

When checking the processing performed
by the linkage editor, refer to the module
map produced by the linkage editor showing
the structure of the load module. The
module map names the modules that have been
incorporated into the program. The
compiler produces an external symbol
dictionary (ESO) listing if requested by
the ESO option. The ESO listing indicates
the external names that the linkage editor
is to resolve in order to create a load
module. The linkage editor is described in
Chapter 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 also listed in the messages
publication for this compiler and
identified by the prefix "IEM". The
messages are always printed on the SYSPRINT
file.

A failure in the execution of a PL/I
program could be caused by one of the following:

- Logical errors in source programs.
- Invalid use of PL/I.
- Unforeseen errors.
- Operating error.
- Invalid input data.
- Unidentified program failure.
- A compiler or library subroutine
  failure.
- System failure.

**Logical Errors in Source Programs**

Logical errors in source programs can often
be difficult to detect. Such errors can
sometimes cause a compiler or library
failure to be suspected. The more common
errors are the failure to convert correctly
from arithmetic data, incorrect arithmetic
operations and string manipulation
operations, and failure to match data lists
with their format lists.

**Invalid Use of PL/I**

It is possible that a misunderstanding of
the language, or the failure to provide the
correct environment for using PL/I, results
in an apparent failure of a PL/I program.
For example, the use of uninitialized
variables, the use of controlled variables
that have not been allocated, reading
records into incorrect structures, the
misuse of array subscripts, the misuse of
pointer variables, conversion errors,
incorrect arithmetic operations, and
incorrect string manipulation operations
can cause this type of failure.

**Unforeseen Errors**

If an error is detected during execution of
a PL/I program in which no on-unit is
provided to terminate execution or attempt
recovery, the job will be terminated.
abnormally. However, the status of a program executed in a batch-processing environment, at the point where the error occurred, can be recorded by the use of an ERROR on-unit that contains the statements:

```pli
ON ERROR BEGIN;
ON ERROR SYSTEM;
PUT DATA;
END;
```

The statement ON ERROR SYSTEM; contained in the on-unit ensures that further errors caused by attempting to transmit uninitialized variables do not result in a permanent loop.

### Operating Error

A job could fail because of an operating error, such as running a job twice so that a data set becomes overwritten or erroneously deleted. Other operating errors include getting card decks into the wrong order and the failure to give operators correct instructions for running a job.

### Invalid Input Data

A program should contain checks to ensure that any incorrect input data is detected before it can cause the program to fail.

Use the COPY option of the GET statement if you wish to check values obtained by stream-oriented input. The values will be listed on the file named in the COPY option. If no file name is given, SYSPRINT is assumed.

### Unidentified Program Failure

In most circumstances, an unidentified program failure should not occur when using the optimizing compiler. Exceptions to this could include the following:

- When the program is executed in conjunction with non-PL/I modules, such as FORTRAN or COBOL.
- When the program obtains, by means of record-oriented transmission, incorrect values for use in label, entry, locator, and file variables.
- Errors in job control statements, particularly in defining data sets.

If execution of a program terminates abnormally without an accompanying PL/I execution-time diagnostic message, it is probable that the error that caused the failure also inhibited the production of a message. In this situation, it is still possible to check the PL/I source program for errors that could result in overwriting areas of the main storage region that contain executable instructions, particularly the communications region, which contains the address tables for the execution-time error-handling routine. These errors may also be present in modules compiled by the checkout compiler with NODIAGNOSE and COMPATIBLE and executed in conjunction with the modules produced by the optimizing compiler. The types of PL/I program that might cause the main storage to be overwritten erroneously are:

- Assignment of a value to a non-existent array element. For example:

```pli
DCL ARRAY(10);
.
.
DO I = 1 TO 100;
ARRAY(I) = VALUE;
```

To detect this type of error in a module compiled by the optimizing compiler, enable the SUBSCRIPTRANGE condition. For each attempt to access an element outside the declared range of subscript values, the SUBSCRIPTRANGE condition will be raised. If there is no on-unit for this condition, a diagnostic message will be printed and the ERROR condition raised. This facility, although expensive in execution time and storage space, is a valuable program-checkout aid.

- The use of incorrect locator values for locator (pointer and offset) variables. This type of error is possible if a locator value is obtained by means of record-oriented transmission. Check that locator values created in a program, transmitted to a data set, and subsequently retrieved for use in another program, are valid for use in the second program.

An error could also be caused by attempting to free a non-based variable. This could be caused by freeing a based variable when its qualifying pointer value has been changed. For example:

```pli
DCL A STATIC,B BASED (P);
ALLOCATE B;
P = ADDR(A);
FREE B;
```

- The use of incorrect values for label,
entry, and file variables. Errors similar to those described above for locator values are possible for label, entry, and file values that are transmitted and subsequently retrieved.

- The use of the SUBSTR pseudovariable to assign a string to a position beyond the maximum length of the target string. For example:

```
DCL X CHAR(3);
I=3;
SUBSTR(X,2,I) = 'ABC';
```

The STRINGRANGE condition can be used to detect this type of error in a module compiled by the optimizing compiler.

**Compiler or Library Subroutine Failure**

If you are absolutely convinced that the failure is caused by a compiler failure or a library subroutine failure, you should notify your management, who will initiate the appropriate action to correct the error. This could mean calling in IBM personnel for programming support to rectify the problem. Before calling IBM for programming support, refer to the instructions for providing the correct information to be used in diagnosing the problem. These instructions are given in Appendix C, "Requirements for Problem Determination and APAR Submission."

Meanwhile, you can attempt to find an alternative way to perform the operation that is causing the trouble. A bypass is often feasible, since the PL/I language frequently provides an alternative method of performing a given operation.

**System Failure**

System failures include machine malfunctions and operating system errors. These failures should be identified to the operator by a system message.

**Statement Numbers and Tracing**

The compiler FLOW option provides a valuable program-checkout aid. The FLOW(n,m) option creates a table of the numbers of the last "n" branch-out and branch-in statements, and the last "m" procedures and on-units to be entered. A "branch-out" statement is a statement that transfers control to a statement other than that which immediately follows it, such as a GOTO statement. A branch-in statement is a statement that receives control from a statement other than that which immediately precedes it, such as a PROCEDURE, ENTRY, or any other labeled statement.) The figure you choose for "n" should be large enough to provide a usable trace of the flow of control through the program. Alternatively, if you do not specify the FLOW option explicitly, defaults for the FLOW option will be used.

The trace table can be obtained by any of the methods described below.

The trace is printed whenever an on-unit with the SNAP option or a PUT ALL statement is encountered. It gives both the statement numbers and the names of the containing procedures or on-units. For example, an ERROR on-unit that results in both the listing of the program variables and the statement number trace can be included in a PL/I program as follows:

```
ON ERROR SNAP BEGIN;
ON ERROR SYSTEM;
PUT DATA;
END;
```

A flow trace can be specified as part of the output from the PL/I dump facility PLIDUMP, discussed later in this chapter.

**Dynamic Checking Facilities**

It is possible for a syntactically-correct program to produce incorrect results without raising any PL/I error conditions. This can be attributed to the use of incorrect logic in the PL/I source program or to invalid input data. Detection of such errors from the resultant output (if any) can be a difficult task. It is sometimes helpful to have a record of each of the values assigned to a variable, particularly label, entry, loop control, and array subscript variables. This can be obtained by using the CHECK prefix option. Note that, unless care is exercised, the indiscriminate use of the facilities described below will result in a flood of unwanted and unusable printout.

A CHECK prefix option can specify program variables in a list. Whenever a variable that has been included in a check-list is assigned a new value, the CHECK condition is raised. The standard system action for the CHECK condition is to print the name and new value of the variable that caused the CHECK condition to be raised. An example of a CHECK prefix options list is:
If the CHECK condition is to be raised for all the variables used in a program, the CHECK prefix option can be more simply specified without a list of items. For example:

```
(CHECK): TEST: PROCEDURE;
```

**Control of Exceptional Conditions**

During execution of a PL/I object program, a number of exceptional conditions can be raised, either as a result of program-defined action, or as a result of exceeding a hardware limitation. PL/I contains facilities for detecting such conditions. These facilities can be used to determine the circumstances of an unexpected interrupt, perform a recovery operation, and permit the program to continue to run. Alternatively, the facilities can be used to detect conditions raised during normal processing, and initiate program-defined actions for the condition. Note that some of the PL/I conditions are enabled by default, some cannot be disabled, and others have to be enabled explicitly in the program. Refer to the language reference manual for this compiler 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 standard system action for the ERROR condition for which there is no on-unit, is, in batched processing, to raise the FINISH condition, and in conversational processing, to give control to the terminal. The FINISH condition is also raised for the following:

- When a SIGNAL FINISH statement is executed.
- When a PL/I program completes execution normally.
- On completion of an ERROR on-unit that does not return control to the PL/I program by means of a GOTO statement.
- When a STOP statement is executed or when an EXIT statement is executed in a major task.

The standard system action for the FINISH condition in batched processing is to terminate the task, and, in conversational processing, to give control to the terminal.

**Use of the PL/I Preprocessor in Program Checkout**

During program checkout, it is often necessary to use a number of the PL/I conditions and the on-units associated with them) and subsequently to remove them from the program when it is found to be satisfactory. The PL/I preprocessor can be used to include 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 compiled for execution.

A standard set of PL/I program checkout statements would include both the enabling of any conditions that are disabled by default and the provision of the appropriate on-units. The %INCLUDE statement that causes the inclusion of the set of program checkout statements would usually be placed after any on-units that must remain in the program permanently in order to cancel their effect during program checkout.

**On-Codes**

On-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 an on-code. You can obtain the on-code by using the condition built-in function ONCODE in the on-unit. The on-codes are described in the language reference manual for this compiler.

**Dumps**

Should the checks given above fail to reveal the cause of the error, it may be necessary to obtain a printout, or dump, of the main storage region 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 also be obtained to determine the machine instructions and data present in main storage when the failure occurred. The use of a hexadecimal storage dump requires a knowledge of assembler language programming and an understanding of object program organization.

Refer to the execution logic manual for this compiler for information about the organization of the object programs produced by the optimizing compiler, and how to interpret a storage dump.

To obtain a formatted PL/I dump, you must invoke the PL/I resident library dump module by calling PLIDUMP. Note that a DD statement with the ddname PLIDUMP or PLLDUMP must be supplied to define the data set for the dump.

The data set defined by the PLIDUMP DD statement must have DSORG=PS specified or assumed by default, and must have one of the following attributes:

- It must be allocated to SYSOUT.
- It must be allocated to the terminal or unit-record device.
- DISP=MOD must be specified.

The page size of the PLIDUMP output is taken from the PAGESIZE field of PLITABS.

PLIDUMP can be invoked 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 can be a character-string expression or a decimal constant with which you can identify the output produced by PLIDUMP. The format of the PLIDUMP statement is:

```
CALL PLIDUMP([ options-list' [,user-identification]));
```

The options-list is a contiguous string of characters that may include the following:

T To request a trace of active procedures, begin blocks, on-units, and library modules.

NT To suppress the output produced by T above.

F To request a complete set of attributes for all files that are open, and the contents of the buffers used by the files.

NF To suppress the output produced by F above.

S To request the termination of the program after the completion of the dump. Note: The FINISH condition is not raised.

C To request continuation of execution after completion of the dump.

H To request a hexadecimal dump of the main storage partition used by the program.

NH To suppress the hexadecimal dump.

B If T is specified, to produce a separate hexadecimal dump of control blocks such as the TCA and the DSA chain that are used in the trace analysis. If F is specified, to produce a separate hexadecimal dump of control blocks used in the file analysis, such as the FCB.

NB To suppress hexadecimal dumps of control blocks.

A To request information relevant to all tasks in a multitasking program.

E To request that an exit be made from the current task of a multitasking program and that execution of the program continues after completion of the requested dump.

O To request information relevant only to the current task in a multitasking program.

The defaults assumed for the above options not specified explicitly are:

T F C A NH NB

The user-identification permits you to specify a character-string expression or a decimal constant to identify individual dumps. It cannot be specified without the preceding argument in the argument list.

Trace Information

Trace information produced by PLIDUMP includes a trace through all the active DSAs. (DSAs will be present for compiled blocks, such as procedures and on-units, and for library routines.) For on-units, the dump gives the values of any condition built-in functions that could be used in the on-unit, regardless of whether the on-unit actually used the condition built-in function. If a hexadecimal dump is also
Return Codes

Meaning

0000
Normal termination.

1- 999
Return codes available for use with PLIRETC.

1000
Code returned if a STOP statement, an EXIT statement, or a CALL PLIDUMP statement with the "E" or "S" option is executed, or if ISASIZE is insufficient. This will be added to any PLIRETC value.

2000
Code returned if ERROR is raised and there is no ERROR or FINISH on-unit containing a GOTO statement. This value will be added to any PLIRETC value.

4000
Code returned if an interrupt occurs in the PL/I error handler or during program initialization.

4004
Code returned if the PRV (pseudo register vector) is too large.

4008
Code returned if PL/I program has no main procedure.

4012
Not enough main storage available.

4020
Code returned if the program is about to enter a permanent wait state.

4024
Code returned if a task in a multitasking program has terminated without use of the PL/I termination routines.

Figure 12-1. Return codes from execution of a PL/I program

PL/I error handler module (IBMCERR) was invoked.

- The addresses of the library module DSAs back to the most recently-used compiled code DSA.

DSAs and the TCA are described in the execution logic manual for this compiler. A table of statement numbers indicating the flow of control through the program is always produced.

File Information

File information produced by PLIDUMP includes the default and declared attributes of all open files, and the contents of all buffers that are accessible to the dump routine. The information is given in BCD notation, and if hexadecimal output is also requested, in hexadecimal notation also. The address and contents of the FCB are then printed.

Hexadecimal Dump

The hexadecimal dump is a dump of the region of main storage containing the program. The dump is given as three columns of printed output. The left-hand and middle columns contain the contents of storage in hexadecimal notation. The third column contains a BCD translation of the first two columns. For hexadecimal characters that cannot be represented by a printable BCD character, a full stop is printed.

Return Codes

Both the compilation and the link editing of a PL/I program will result in a return code being passed to indicate the severity of any errors found. It is possible to pass a return code from a PL/I program, either for examination in a subsequent job step if execution of that step is conditional upon the value of the code returned, or merely to indicate conditions that were encountered during execution. Conditional execution of a job step is determined by use of the COND parameter of the JOB or EXEC statement.

Return codes can be set in a PL/I program by passing as an argument to the CALL PLIRETC statement a code represented as a variable with the attributes FIXED.
BINARY(31,0). The range of codes used 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 higher than 999 are returned if an error causes the program to terminate. In other cases the return code for the program will be added to any code created by use of the CALL PLIRETC statement. In other cases it will overwrite any code set by use of the CALL PLIRETC statement. Other error situations, listed in Figure 12-1, will also cause a program-generated return code to be overwritten.

If a return code in the 4000-4024 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 C, "Problem Determination and APAR Submission", describes the materials that will be required for examination by IBM in such circumstances.

The ABEND Facility

Standard system action when the ERROR condition is raised and there is either no ERROR on-unit, or an ERROR on-unit that terminates normally (that is, one that does not contain a GOTO STATEMENT) is to terminate without issuing an ABEND. If the standard module IBMBEER in the PL/I resident library is altered as indicated below then the ABEND facility becomes available. IBMBEER will be called whenever the PL/I program is to be terminated as a result of the ERROR condition having been raised.

Alter the Standard Module IBMBEER

A non-zero return should be set in register 15. The program termination modules IBMTPIT and IEMTPIT call IBMBEER and interrogate register 15 on return; a non-zero value causes an ABEND to be issued. The value in register 15 is passed as a parameter to the ABEND which allows the options of user completion code, dump, and STEP ABEND to be selected.

When IBMBEER is called, the content of register 1 will be set as follows to signify whether or not there is an ERROR on-unit:

- positive - ERROR on-unit
- negative - no ERROR on-unit

An example of a user-written IBMBEER module is shown in Figure 12-2.

```
ENTRY IBMBEERA DEFINE ENTRY POINT
USING *,15 ADDRESSABILITY

IBMBEERA EQU * ENTRY POINT
| LTR R1,R1 BRANCH
| BNM ERRB IF ON-UNIT
| L 15,RETNCOD1 SET RETURN CODE NON-ZERO
| BR 14 RETURN
| ERRB EQU *
| L 15,RETNCOD2
| BR 14 *
| DUMP EQU 128 128=DUMP, 0=NO DUMP
| STEP EQU 64 64=STEP ABEND, 0=TASK ABEND
| USERCOD1 EQU 3001 USER COMPLETION CODES (MUST BE GREATER THAN 0 AND LESS THAN 4096)
| USERCOD2 EQU 3002
| RETNCOD1 DC AL1(DUMP+STEP)
| DC AL3(USERCOD1)
| RETNCOD2 DC AL1(DUMP+STEP)
| DC AL3(USERCOD2)
END
```

Figure 12-2. Typical User-Written IBMBEER Module
Chapter 13: Linking PL/I and Assembler-Language Modules

This chapter describes how to create programs that combine routines written in PL/I and assembler language. It explains how a PL/I program invokes an assembler-language routine and, conversely, how an assembler-language routine invokes a PL/I procedure.

Before describing any of the linkages in detail, the chapter discusses the PL/I environment that must be preserved when invoking an assembler-language routine from PL/I, and which must be created when invoking a PL/I procedure from an assembler-language routine.

The PL/I Environment

The PL/I environment is the term used to describe a number of control blocks created by routines that are provided by the OS PL/I Resident and Transient Libraries to satisfy the storage-management and error-handling requirements of a PL/I procedure.

When a PL/I program invokes an assembler-language routine, the invoked routine must ensure that the PL/I environment is preserved. The PL/I environment is preserved by observing the standard IBM System/360 linkage conventions, which include the storing of register values in a save area, and by ensuring that the content of register 12 is not modified by the assembler routine if PL/I is to handle interrupts that occur during execution of the assembler routine. Register 13 must be set to the address of a new save area established by the assembler routine.

ESTABLISHING THE PL/I ENVIRONMENT

The PL/I environment is established by the OS PL/I Resident Library routine IBMBPIR and the OS PL/I Transient Library routine IBMTPIR for a non-multitasking program and by IBMTPIR and IBMTPII for a multitasking program. An assembler-language routine that invokes a PL/I procedure for which the PL/I environment has not been established can use one of three standard entry points to establish the environment. The routine IBMBPIR or IBMTPIR (with IBMBPII or IBMTPII) is entered through a control section which has three standard entry points, PLISTART, PLICALLA, and PLICALLB; these are described later in this chapter.

Use of PLIMAIN to Invoke a PL/I Procedure

Once IBMBPIR or IBMTPIR (with IBMBPII or IBMTPII) has created the environment, it transfers control to the PL/I procedure whose address is contained in the compiler-generated control section PLIMAIN. Normally, after link editing, PLIMAIN will contain the entry point address of the first, or only, PL/I main procedure in the program. 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 PLIMAIN the address of the entry point of the procedure it is to invoke. The example in Figure 13-1 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 13-2 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 OS PL/I Resident Library.

Once the PL/I environment has been established, it can, as shown in the example in Figure 13-3, be preserved, and any PL/I procedure can be invoked subsequently by loading the address of its entry point into register 15, and executing a branch-and-link-register instruction to it.

PLISTART, PLICALLA, AND PLICALLB

PLISTART: PLISTART is 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, or, alternatively, as much as is specified. This entry point is normally used when the PL/I procedure is invoked directly from the operating system. It enables an assembler routine to pass to
ENTRY PLIMAIN
IA 1, ARGLIST
L 2,=V(PLIMAIN)  INSERT ADDRESS IN PLIMAIN
L 3,=V(MYPROG)  OF ENTRY TO
ST 3,0(2)       MYPROG
L 15,=V(PLICALLA)
BALR 14,15
.
.
ARGLIST DC A(arg1) FIRST ARGUMENT PASSED TO MYPROG
DC X'80'       LAST ARGUMENT PASSED TO MYPROG
DC AL3(arg2)
PLICALLA: PLICALLA is 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 used when the PL/I environment must be established for a PL/I procedure that can use for its dynamic storage only a specified amount of the available storage. PLICALLB can optionally specify where that storage is to begin.

Further details and examples using PLISTART, 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 should take the following action to preserve the PL/I environment:

- If the assembler routine is to use the PL/I error-handler, 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 two bytes 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.
- If the assembler routine is not to use the PL/I error-handler and does not invoke a further function routine, the SPIE macro must be used to reset the
EXEC HASMHC,PARM.ASM='LOAD,NODECK'

**MYPROG**

**CSECT**

ENTRY ASSEM

ST M 14,12,12(13)

BALR 10,0

USING *,10

LA 4,SAVEAREA

ST 13,4(4)

ST 4,8(13)

LR 13,4

* SR 1,1

SET REGISTER 1 TO ZERO WHEN
A PARAMETERLESS ENTRY POINT TO
PROCEDURE THAT DOES NOT RETURN
A VALUE IS TO BE INVOKED

* L 15,=V(PLICALLA)

CALL THE PL/I PROCEDURE THAT
HAS OPTIONS(MAIN) AND SO SET
UP THE PL/I ENVIRONMENT AND
THEN CALL ASSEM.

* L 13,4(13)

ON RETURNING FROM PL/I

L 14,12(13)

RESTORE REGISTERS

LM 1,12,24(13)

AND

BR 14

SET FLAG 1 TO ZERO WHEN
A PARAMETERLESS ENTRY POINT TO
PROCEDURE THAT DOES NOT RETURN
A VALUE IS TO BE INVOKED

* DC 'C'ASSEM'

THE NAME IN PL/I FORMAT

DC ALL(5)

ASSEM

EQU *

ST M 14,12,12(13)

BALR 10,0

USING *,10

* LA 0,104

GET STORAGE FOR A NEW DSA

L 1,76(13)

ADDRESS OF START OF CURRENTLY-

AVAILAble Storage

ALR 0,1

IS THERE ENOUGH SPACE LEFT?

CL 0,12(12)

ENOUGH

ENOUGH

L 15,16(12)

LOAD ADDR. OF OVERFLOW ROUTINE

BALR 14,15

AND BRANCH TO IT.

* ST 0,76(1)

STORE ADDRESS OF START OF
REMAINING AVAILABLE STORAGE

IN NEW DSA AT OFFSET 76

* ST 13,4(1)

SET BACK CHAIN

ST 1,8(13)

SET FORWARD CHAIN

MVC 72(4,1),72(13)

COPY ADDRESS OF WORKSPACE FOR
USE BY THE PL/I LIBRARY

LR 13,1

POINT 13 AT NEW DSA

MVI 0(13),X'80'

SET FLAGS IN THE DSA TO

MVI 1(13),X'00'

ERROR-HANDLING

MVI 86(13),X'91'

IN THE ASSEMBLER ROUTINE

MVI 87(13),X'CO'

* SR 5,5

R5 MUST BE ZERO WHEN CALLING
AN EXTERNAL PL/I PROCEDURE.

* SR 1,1

SET REGISTER 1 TO ZERO WHEN
A PARAMETERLESS ENTRY POINT TO
PROCEDURE THAT DOES NOT RETURN
A VALUE IS TO BE INVOKED

Figure 13-3. (Part 1 of 2). Invoking PL/I procedures from an assembler routine

Chapter 13: Linking PL/I and Assembler-Language Modules 167
/*
   L 15,=V(HEAD)          CALL PL/I TO 'HEAD' PAGE
   BALR 14,15

   LOOP EQU *
   LA 1,ARGVLIST1
   L 15,=V(PLIN)          CALL PL/I TO READ AND ADD
   BALR 14,15

   L 3,RESULT
   LTR 3,3
   EM OUTLOOP             TEST RESULT AND
   BR LOOP                BRANCH OUT IF IT IS NEGATIVE.

   LA 1,ARGVLIST2
   L 15,=V(PLOUT)         CALL PL/I TO OUTPUT RESULT
   BALR 14,15
   B LOOP

   OUTLOOP EQU *
   SR 1,1                  SET REGISTER 1 TO ZERO
   L 15,=V(FOOT)          CALL PL/I TO 'FOOT' PAGE
   BALR 14,15

   L 13,4(13)              RETURN TO THE PL/I PROC WITH
   LM 14,12,12(13)         OPTIONS(MAIN).
   BR 14

   ARGVLIST1 DC A(DATA)
   ARGVLIST2 DC X'80'
   DATA DC F'123'
   RESULT DC F'0'
   SAVEAREA DC 18F'0'
   END MYPROG

   /*
    //STEP2 EXEC PLIXCLG
    //PLI.SYSIN DD *
    * PROCESS;
    | MAIN: PROC OPTIONS(MAIN);
    | DCL ASSEM ENTRY;
    | CALL ASSEM;
    | END;
    * PROCESS;
    | PLIN: PROC(I) RETURNS(FIXED BIN(31));
    | DCL (I,J) FIXED BIN(31);
    | GET LIST(J);
    | RETURN(I+J);
    | HEAD: ENTRY;
           PUT LIST('THE FIRST LINE OF OUTPUT AT THE TOP OF THE PAGE')
           PAGE;
           PUT SKIP(2);
           END;
    * PROCESS;
    | PLOUT: PROC(K);
    | DCL K FIXED BIN(31);
    | PUT LIST(K);
    | RETURN;
    | FOOT: ENTRY;
           PUT LIST('END OF THE OUTPUT FOR THIS JOB') SKIP(2);
           END;
    */
    //GO.SYSIN DD *
    50 77 123 234 345 456 -23 -100 -123 -234
    */

Figure 13-3. (Part 2 of 2). Invoking PL/I procedures from an assembler routine
Calling Assembler Routines from PL/I

INVOVING A NON-RECURSIVE AND NON-REENTRANT ASSEMBLER ROUTINE

When a PL/I program invokes a non-recursive and non-reentrant assembler-language routine, the assembler-language routine must follow System/360 linkage conventions and save the registers for use by PL/I on return from the assembler-language routine. The register values are stored in the PL/I DSA, the address of which is contained in register 13 on entry to the assembler-language routine. This address must then be stored in the backchain word in a save area defined by the assembler routine itself. The appropriate assembler instructions should be executed immediately the assembler routine is invoked in order to achieve the given objectives. Before returning to the PL/I routine, the assembler routine must restore the registers to the values held when the PL/I routine invoked the assembler routine. The example in Figure 13-4 assumes that the assembler routine uses register 10 as its base register.

INVOVING A RECURSIVE OR REENTRANT ASSEMBLER ROUTINE

A recursive or reentrant assembler routine invoked from PL/I 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.

A DSA established by the assembler routine must have its first two bytes set to X'00' if it is to handle any program interrupts. Such a DSA must be at least 80 bytes in length to accommodate both the save area and two fullwords required by PL/I for its housekeeping. If the PL/I error-handler is to service any program interrupts in the assembler-language routine, the DSA should be at least 88 bytes in length. The first byte of the DSA should be set to X'80', the second byte set to X'00', and bytes 87 and 88 (the PL/I error-handler enable cells) set to X'9100'. In addition, a DSA can be as long as is needed to store any values that are to be preserved for use by a particular invocation. Note that a DSA obtained in this way must be a multiple of 8 bytes in length.

Termination of a recursive or reentrant assembler-language routine will release its DSA and cause control to be returned to the invoking routine.

The example in Figure 13-5 shows how to create and release a DSA in a recursive or reentrant assembler routine.

USE OF REGISTER 12

If an assembler routine that modifies register 12 is to be invoked by a PL/I
Figure 13-5. Invoking a recursive or reentrant assembler routine

Calling PL/I Procedures from Assembler Language

The simplest way to invoke a single external PL/I procedure from an assembler-language routine is to give the PL/I procedure the MAIN option and invoke it using entry point PLICALLA. All that is required is to load the address of PLICALLA into register 15 and then to branch and link to it. When PLICALLA is used in this way, the PL/I environment is created and control is then passed by way of PLIMAIN to the first (or only) main PL/I procedure in the program. Use of this technique will cause the PL/I environment to be established separately for each invocation.

If the 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 invokes the assembler routine.

procedure, any program-check interrupts will result in an unpredictable program failure unless the routine establishes its own error handling for program-check interrupts. Consequently, the routine should be amended to use a register other than register 12 so that the PL/I error-handling can be used, or it can issue a supervisor SPIE or STAE macro to establish its own program interrupt or abnormal termination handling facilities. The routine must subsequently restore PL/I error-handling facilities before returning to PL/I. This is discussed further in "Overriding and Restoring PL/I Error-handling in an Assembler-language Routine" later in this chapter. (A routine that changes the content of register 12 should also store it on entry and restore it on return.)
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 13-3 contains an assembler-language routine that establishes the PL/I environment once only for multiple invocations of PL/I procedures.

In this example, the assembler routine MYPROG receives control initially from the supervisor, and invokes the PL/I main procedure MAIN using the entry point PLICALLA to the PL/I initialization routine. The PL/I procedure MAIN immediately reinvokes the same assembler routine at the entry point ASSEM. Note that, in this example, this name must be an odd number of characters to ensure that the next instruction is halfword aligned. At this entry point, the PL/I environment is stored, and a new DSA, 104 bytes in length, is created in a manner similar to that previously given for creating a DSA in a recursive or reentrant assembler-language routine. If there is insufficient room for the new DSA, the PL/I overflow routine is invoked to attempt to obtain storage for the DSA elsewhere in storage.

The instructions in the assembler routine following the label ENOUGH through to the instruction that loads the address of the PL/I entry point HEAD are concerned with setting up the DSA so that the correct environment exists when the routine invokes the external PL/I procedures PLIN and PLOUT and the secondary entry points within them. These instructions should always be present in order to preserve the PL/I environment set up by the main procedure for subsequent use by any assembler-invoked PL/I procedures.

Note that when an external PL/I procedure is invoked, register 5 must be set to zero, and that a PL/I procedure, such as PLIN in this example, that returns a value will assign the value to the last address in the argument list, ARGTLST1. This address is the address of the assembler-defined storage for RESULT. The constant X'80' in the first byte of the fullword containing the address of RESULTS in ARGTLST1 indicates that it is the last fullword in the argument list.

If an assembler-language routine invokes a PL/I procedure without passing any parameters to it and without expecting any value to be returned from it, register 1 must be set to zero. In this example, the procedure PLIN contains a RETURN (expression) statement, but when invoked through the parameterless entry point HEAD, does not return a value to the invoking routine. Similarly, the procedure PLOUT contains the parameterless entry point FOOT and does not return a value.

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 PLISTART, PLICALL A, or PLICALL B to invoke the PL/I initialization routines. The three entry points are used as follows:

For PLISTART, the assembler-language routine must insert in register 1 the address of a fullword which in turn contains the address of a halfword prefix to a character string. The character string, which must start on a fullword boundary, can contain a parameter string similar to that which can be specified in the PARM field of a JCL EXEC statement; for example, 'ISASIZE(4K),R/INPUT'. The halfword prefix must contain the number of characters in the string. This entry point is useful when a PL/I routine is "attached" by an assembler routine, because the entry point of the PL/I routine does not have to be changed. The use of PLISTART is illustrated in Figures 13-6 and 13-7.

For PLICALL A, the assembler-language routine must insert in register 1 the address of the argument list that contains the addresses of any arguments to be passed to the PL/I procedure.

For PLICALL B, the assembler-language routine must insert in register 1 the address of an argument list that contains the following:

- The address of the argument list containing addresses to be passed to PL/I, and optionally,
- The address of the length of storage to be made available to the program in a non-multitasking program or the major task in a multitasking program. The default for this length is half the available storage for a non-multitasking program or 8K bytes for the major task of a multitasking program. The length of the initial storage area (ISA) passed must be a multiple of eight bytes, so that the ISA both starts and ends on a double-word boundary.
- The start address of the initial storage area (ISA) to be used by the PL/I program. This storage must be aligned on a double word. For further information, refer to the discussion of the ISASIZE option in Chapter 4.
LA 1,PLISTHWD GET PLIST ADDRESS
ATTACH EP=PLIPROG ATTACH PL/I PROGRAM
*
PLISTHWD DS OF
DC X'80' FLAG LAST WORD OF PLIST
DC AL3(PLISTHWD)
PLISTHW DC AL2(L'PLISTCH) LENGTH OF PARM STRING
PLISTCH DC 'ISASIZE(8K),R/INPUT' PARM DATA

Figure 13-6. Use of PLISTART for ATTACH

LA 1,PLISTHWD GET PLIST ADDRESS
L 15,=V(PLISTART) GET PL/I ENTRY POINT
BALR 14,15 CALL PL/I ROUTINE
*
*
DS OF
PLISTHWD DC X'80' FLAG LAST WORD OF PLIST
DC AL3(PLISTHWD)
PLISTHW DC H'0' NULL PARM STRING
PLISTCH DC AL2(0)

Figure 13-7. Use of PLISTART passing null parameter string

<table>
<thead>
<tr>
<th>Option</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>REPORT</td>
<td>X'80' in first byte</td>
</tr>
<tr>
<td>NOREPORT</td>
<td>X'40' in first byte</td>
</tr>
<tr>
<td>SPIE</td>
<td>X'20' in first byte</td>
</tr>
<tr>
<td>NOSPIE</td>
<td>X'10' in first byte</td>
</tr>
<tr>
<td>STAE</td>
<td>X'08' in first byte</td>
</tr>
<tr>
<td>NOSTAE</td>
<td>X'04' in first byte</td>
</tr>
<tr>
<td>COUNT</td>
<td>X'02' in first byte</td>
</tr>
<tr>
<td>NOCOUNT</td>
<td>X'01' in first byte</td>
</tr>
<tr>
<td>FLOW</td>
<td>X'80' in second byte</td>
</tr>
<tr>
<td>NOFLOW</td>
<td>X'40' in second byte</td>
</tr>
</tbody>
</table>

Figure 13-8. Coding the options word

- The address of the length of storage to be made available to each of the subtasks in a multitasking program. The default for this length is 8K bytes for each subtask. This value is ignored for a non-multitasking program. The length of the ISA must be a multiple of eight bytes.

- The address of the maximum number of concurrent subtasks that can be attached at any one time. This value is ignored in a non-multitasking program. The default for this value is 20.

- The address of the options word, in which the execution-time options for a program compiled by the optimizing

LA 1,ARGLIST
L 15,=V(PLICALLA)
BALR 14,15
*
*
ARGLIST DC A(arg1) ADDRESS OF FIRST ARGUMENT PASSED TO PL/I
DC A(arg2) ADDRESS OF SECOND ARGUMENT PASSED TO PL/I
*
*
DC X'80' END OF ARGUMENT LIST FLAG
DC AL3(argn or return-value) ADDRESS OF LAST ARGUMENT OR RETURNED VALUE

Figure 13-9. Use of PLICALLA
LA 1,ALIST
L 15,=V(PLICALLB)
BALR 14,15

ALIST    DC A(ARGLIST) ADDRESS OF ARGUMENT LIST
        DC A(LENGTH) LENGTH OF STORAGE FOR PL/I
        *    DC A(ISA) ADDRESS OF ISA
        DC A(0) TASK ISA - NOT USED
        DC A(0) NUMBER OF CONCURRENT SUITASKS - NONE
        DC X'80' END OF ARGUMENT LIST FLAG
        DC AL3(OPTIONS) OPTIONS WORD
ARGLIST   DC A(arg1) ADDRESS OF FIRST ARGUMENT
        DC A(arg2) ADDRESS OF SECOND ARGUMENT
        DC X'80' END OF ARGUMENT LIST FLAG
        DC AL3(argn or return-value) ADDRESS OF LAST ARGUMENT
        *    LENGTH DC F'8192' ROUTINE'S STORAGE LIMITED TO 8 K BYTES
        ISA DS 1024D ROUTINES ISA STARTS HERE
        OPTIONS DC X'84000000' REPORT AND NOSTAE REQUESTED

Figure 13-10. Use of PLICALLB

compiler are specified. These options are: REPORT; STAE; SPIE; COUNT; and FLOW. They are described in Chapter 4. The hexadecimal value for each option is given in Figure 13-8.

Note that the first byte in the last address word in each of these argument lists must contain X'80'. The examples in Figures 13-9 and 13-10 show the use of PLICALLA and PLICALLB to invoke the first (or only) main PL/I procedure in the program. The PL/I programs in these cases do not perform multitasking.

If it is necessary to reestablish the PL/I environment for each invocation of a PL/I procedure that is not the first (or only) main procedure in the program, the user of either entry point PLICALLA or PLICALLB must insert in PLIMAIN the address of the appropriate entry point to the required PL/I procedure. The example in Figure 13-1 sets the address in PLIMAIN to that of the external entry name MYPROG.

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 must contain the address of the PL/I routine to be invoked. Figure 13-2 shows how this is inserted.

PL/I Calling Assembler Calling PL/I

The information given in the preceding sections should be sufficient to write programs that include a PL/I procedure that invokes an assembler-language routine that invokes a further PL/I procedure. Figure 13-3 contains an example of a program that performs this type of processing.

Assembler Calling PL/I Calling
Assembler

The information given in the preceding sections should be sufficient to write programs that include an assembler-language routine that invokes a PL/I procedure that in turn invokes an assembler-language routine. Figure 13-3 contains an example of a program that performs this type of processing.

Overriding and Restoring PL/I Error-handling

An assembler-language routine invoked from PL/I can override PL/I error-handling by issuing its own SPIE macro to handle program interrupts or STAE macro to handle abnormal terminations. If the SPIE macro is issued, the address of the PL/I PICA must be saved. A routine that cancels PL/I error-handling must restore the PL/I error-
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
STAE (operands) ESTABLISH NEW ABEND HANDLER
SPIE (operands) ESTABLISH INTERRUPT HANDLER
ST 1,SAVESPIE STORE OLD PICA ADDRESS

ASSEM

STM 14,12,12(13) STORE PL/I ENVIRONMENT
BALR 10,0 ESTABLISH BASE REGISTER
USING *,10
STAE (operands) ESTABLISH NEW ABEND HANDLER
SPIE (operands) ESTABLISH INTERRUPT HANDLER
ST 1,SAVESPIE STORE OLD PICA ADDRESS

. . .

STAE 0 RESTORE PL/I ERROR HANDLING
L 1,SAVESPIE RESTORE PICA ADDRESS
SPIE MF=(E,(1))
L 13,4(0,13) RESTORE PL/I ENVIRONMENT
LM 14,12,12(13) RETURN TO PL/I
BR 14

SAVESPIE DS A

Figure 13-11. Method of overriding and restoring PL/I error-handling

Arguments and Parameters

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.

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

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, for example, an array is passed from PL/I to an assembler routine, the address in the parameter list is that of the first element of the array.

Note that if a particular data item is not byte-aligned (for example, an unaligned bit string), the address in the parameter list is that of the byte that contains the start of the data item. Also, varying length character and bit strings are preceded in storage by a two-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 either the address of a data item or the address of a control block, depending on the type of data that is being passed.

For arithmetic element variables, the address in the parameter list is that of the variable itself. For all other problem data types, the address in the parameter list is that of a control block. The formats of locator/descriptors and of control blocks for program control data are given in the execution logic manual for this compiler.

It is recommended that the use of this type of linkage is avoided wherever possible. Access to locator descriptors is normally only necessary when the full attributes of the arguments are not known by the assembler routine. The use of function references (which cannot be used with the ASSEMBLER option) can be avoided by passing the receiving field as a parameter to the assembler routine.

PASSING ARGUMENTS FROM AN ASSEMBLER-LANGUAGE ROUTINE

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'00' in its first byte. If the PL/I routine executes a RETURN(expression) statement, the last address in the parameter list must be that of the field to which PL/I is to assign the returned value.

Each address in the parameter list must be either the address of a data item or the address of a control block that describes a data item, depending upon the type of data that is being passed. For arithmetic element variables, the address in the parameter list must be that of the variable itself. For all other problem data types, the address in the parameter list must be that of a locator/descriptor. For program control data, the address in the parameter list must be that of a control block. The formats of locator/descriptors and of control blocks for program control data are given in the execution logic manual for this compiler.

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 in 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.
If you intend to use the PL/I sort facilities, the version of OS generated for your installation must include either a copy of the OS type 1 sort/merge program (Program Number 360S-SM-023) or a copy of the OS program product sort/merge program (Program Number 5734 SML). The PL/I sort facilities make use of either OS sort/merge program to arrange records according to a predetermined sequence.

Note: If any of the data sets used by the sort program are to reside on an IBM 3330 or IBM 3333 device, the OS program product sort/merge program (Program Number 5734 SML) must be used.

The sort/merge program includes exit points to enable user-written routines to be entered at particular stages during the sorting operation and which provide access to records that are being sorted. The PL/I sort facilities provide an interface to enable the sort/merge 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. It should be used in conjunction with the relevant OS Sort/Merge publication. The type 1 program is described in OS Sort/Merge; the program product is described in the publications OS Sort/Merge: Programmer's Guide and OS/VSE Sort/Merge: Programmer's Guide. These books are referred to collectively throughout the remainder of this chapter as "the sort/merge publication".

Storage Requirements

The minimum storage requirements for the sort program when used in conjunction with a PL/I program compiled by the optimizing compiler is 12000 bytes or 26000 bytes in an MVT environment. Additional storage requirements exist if the sort program handles records that are greater than 400 bytes in length and if it uses direct-access devices for input, output, or intermediate storage. Efficiency is enhanced if additional main storage can be provided. Refer to the sort/merge publication for further information.

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 subroutine provided by the OS 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 subroutine has entry points for four types of processing, shown in Figure 14-1.

<table>
<thead>
<tr>
<th>Entry Point</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLISRTA</td>
<td>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).</td>
</tr>
<tr>
<td>PLISRTB</td>
<td>Invokes the sort/merge program and specifies the use of user exit E15. A PL/I procedure invoked at user exit E15 will supply all the records to be sorted. The sorted records are written directly onto the data set SORTOUT.</td>
</tr>
<tr>
<td>PLISRTC</td>
<td>Invokes the sort/merge program and specifies the use of user exit E35. The sort/merge program retrieves records from the data set SORTIN. The sorted records are passed to a PL/I procedure invoked at user exit E35. This procedure will handle any output that is required.</td>
</tr>
<tr>
<td>PLISRTD</td>
<td>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.</td>
</tr>
</tbody>
</table>

Figure 14-1. Sort/merge program entry points

After completion of the sort, the sort/merge program passes a return code to the invoking program to indicate whether
the sort is successful or not. The
invoking procedure must include a variable
with the attributes FIXED BINARY(31) to
receive this return code, and the name of
the variable must always be included in the
argument list of the CALL statement that
invokes sort/merge. The return codes and
their meanings, are:

0  Sort successful
16  Sort unsuccessful

PROcedures Invoked By Way of Sort User
Exits

Both external and internal PL/I procedures
can be invoked by way of sort user exits.
The use of external PL/I procedures should
present no problems so long as their entry
names are 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.
A PL/I procedure invoked by way of user
exit E35 must include a character-string
parameter; a PL/I procedure invoked from
user exit E15 must pass a record to the
sort/merge program by means of a RETURN
statement with a character-string
expression as its argument.

Varying-length character strings can be
returned from an E15 exit procedure and
sorted as variable-length records.

Varying-length character strings cannot
be received as parameters in an E35 exit
procedure. However, a variable-length
record passed to an E35 exit procedure can
be declared as an adjustable-length
character string. For example:

E35X: PROC (VREC);
    DCL VREC CHAR(*);

A sorting operation can also be
specified to handle fixed-length records
when the PL/I procedure is to pass varying-
length character strings to it. In this
case, the strings are converted to
fixed-length records of maximum length by
having blanks added to them where
necessary.

Similarly, fixed-length character
strings passed from a PL/I procedure can be
converted into variable-length records and
sorted.

A PL/I procedure invoked by way of 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
subroutine PLIRETC as follows:

CALL PLIRETC(n);

where "n" can have one of the following
values to specify the action required:

For procedures invoked by means of 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. (OS
program product sort/merge program
only.)

For procedures invoked by means of user
exit E35:

4  Pass the next sorted record to the
E35 procedure.
8  Do not return to this procedure.
16  Stop the sort and return immediately
to the invoking procedure. (OS
program product sort/merge program
only.)

Data Sets Used by Sort/Merge

The execution step for a PL/I program that
uses PL/I sort requires job control DD
statements for some or all of the following
data sets in addition to those required by
the PL/I program.

Input Data Sets

If the sort/merge program is to read the
records to be sorted from a data set,
include a DD statement for the data set,
using the ddname SORTIN.
<table>
<thead>
<tr>
<th>Arg.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>arg1</td>
<td>A character string containing the sort/merge SORT statement; this statement must be preceded and followed by a blank character.</td>
</tr>
<tr>
<td>arg2</td>
<td>A character string containing the sort/merge RECORD statement; this statement must be preceded and followed by a blank character.</td>
</tr>
<tr>
<td>arg3</td>
<td>Amount of main storage for the sort/merge program.</td>
</tr>
<tr>
<td>arg4</td>
<td>Name of the variable in the invoking procedure that is to receive the sort return code.</td>
</tr>
<tr>
<td>arg5</td>
<td>Entry point name of the PL/I procedure to be invoked from user exit E15.</td>
</tr>
<tr>
<td>arg6</td>
<td>Entry point name of the PL/I procedure to be invoked from user exit E3S.</td>
</tr>
<tr>
<td>arg7</td>
<td>Replacement ddname characters (see the section &quot;Multiple Invocations of Sort/Merge,&quot; later in this chapter).</td>
</tr>
<tr>
<td>arg8</td>
<td>Diagnostic message listing option (see the section &quot;Sort/Merge Message Listing Options,&quot; later in this chapter).</td>
</tr>
<tr>
<td>arg9</td>
<td>Sorting technique option (see the section &quot;Sort/Merge Sorting Techniques,&quot; later in this chapter).</td>
</tr>
</tbody>
</table>

Figure 14-2. Arguments used when invoking sort/merge

### Output Data Sets

If the sort/merge program is to write the sorted records onto an output data set, include a DD statement for the data set, using the ddname SORTOUT.

### Other Data Sets

For the sort program to execute successfully, it must have access to the following data sets:

- **SORTLIB** The system sort/merge program library.
- **SYSOUT** For sort/merge program diagnostic messages.

The following data sets are needed if the associated facility is to be used:

- **SORTCKPT** If the sort/merge program is to make use of the checkpoint/restart facility.
- **SYSUDUMP** For dumps of main storage if required for debugging the sort program.
- **PLIDUMP** For dumps of main storage if required for debugging the PL/I program.

### 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 by way of the SYSIN input stream.) The control statements are described in the sort/merge publication. Note that the MERGE statement cannot be used when invoking the sort/merge program through the PL/I sort interface. The general syntax of the CALL statement for each of the four entry points is:

### Work Data Sets

The sort/merge program requires at least three magnetic-tape 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 32 data sets. Provide a DD statement for each work data set, using the ddnames SORTWK01 to SORTWK32.
MSORT: PROC OPTIONS(MAIN);

    /* INVOKE THE SORT PROGRAM FOR THE FIRST TIME */
    CALL PLISRTA(' SORT FIELDS=(7,74,CH,A)',
                 ' RECORD TYPE=F,LENGTH=(80)',
                 100000,
                 RETURN_CODE);

    /* INVOKE THE SORT PROGRAM FOR THE SECOND TIME */
    CALL PLISRTA(' SORT FIELDS=(7,74,CH,A)',
                 ' RECORD TYPE=F,LENGTH=(80)',
                 100000,
                 RETURN_CODE,
                 'TASK');

END MSORT;

Figure 14-3. Multiple invocations of sort/merge

The arguments are described in Figure 14-2.

Arguments arg7, arg8, and arg9 are optional. If an optional argument is not used, it need not be specified unless another argument that follows it in the given order is specified. In this case, the unused argument must be specified as a null string. The following sections describe how to use the optional arguments.

Multiple Invocations of Sort/Merge

For multiple invocations of the sort/merge program from a single job step, the standard ddnames of sort data sets (SORTIN, SORTOUT, SORTWK, and SORTCKPT) can be changed by replacing up to the first four characters of the ddnames with a similar number of different characters. This is achieved by specifying the optional argument arg7 in the CALL statements that invoke the sort/merge program. For the invocations of the sort/merge program using modified ddnames, the optional argument should be a character string that contains the replacement characters. Note that the first character of the replacement string must be alphabetic.

For the invocation using the standard sort/merge data sets, arg7 need not be specified unless arg8 is specified, when arg7 should be specified as a null string.

An example of multiple invocation is given in Figure 14-3.

In this example, the first invocation of the sort/merge program requires DD statements with the following ddnames:

    //SORTIN DD ...  
    //SORTOUT DD ...  
    //SORTWK01 DD ...  

The second invocation of the sort/merge program requires DD statements with the following ddnames:

    //TASKIN DD ...  
    //TASKOUT DD ...  
    //TASKWK01 DD ...  

180 OS PL/I Optimizing Compiler: Programmer's Guide
Sort/Merge Message Listing Options

It is possible to select one of five options for specifying how the sort/merge program diagnostic messages are to be produced. The selected option can be specified as an optional argument (arg 8) to the entry point used. The optional argument should contain a character string selected from those given in Figure 14-4.

- **NO** No messages to be printed
- **AP** All messages to be printed on the printer
- **AC** All messages to be printed on the system console
- **CP** Critical messages only to be printed on the printer
- **CC** Critical messages only to be printed on the system console

**Figure 14-4. Sort/merge message listing options**

If no sort/merge listing option is specified, the diagnostic messages will be printed in the way specified when the sort/merge program was generated.

An example of a CALL PLISRTA statement that neither modifies sort/merge ddnames nor specifies a sort/merge message listing option but that does specify a sorting technique is given in Figure 14-6.

```pli
CALL PLISRTA ("SORT FIELDS=(7,74,CH,A)", 'RECORD TYPE=F,LENGTH=(80)", 100000, RETURN_CODE, "'", /* NULL DDNAME ARGUMENT */ 'POLY');
```

**Figure 14-6. Specifying a sort/merge sorting technique option**

Sort/Merge Sorting Techniques

It is possible to select one of four sorting techniques for use by the sort/merge program. The techniques are described in the sort/merge program publication. The selected technique must be specified in an optional argument (arg 9) to the entry point used. The optional argument should contain one of the character strings BALN, CRCX, OSCL, or POLY according to the technique that is required.

If no sorting technique option is specified, the sort/merge program will use the method that is most efficient for the particular job. Specifying that a particular technique is to be used could therefore cause the sort/merge operation to be performed less efficiently.

An example of a CALL PLISRTA statement that does not modify the sort/merge ddnames but that does specify a sort/merge message listing option is given in Figure 14-5.

```pli
CALL PLISRTA ('SORT FIELDS=C7,74,CH,A) , RECORD TYPE=F,LENGTH=(80) , 100000, RETURN_CODE, /* NULL DDNAME ARGUMENT */ 'CP');
```

**Figure 14-5. Specifying a sort/merge message listing option**

Examples of Using PL/I Sort

SORTING RECORDS DIRECTLY FROM ONE DATA SET TO ANOTHER (PLISRTA)

The example in Figure 14-7 illustrates the use of entry point PLISRTA to retrieve records from an input data set (SORTIN), sort them, and write them directly in sorted sequence onto an output data set (SORTOUT).

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 for output, and three direct-access storage extents on a single disk storage drive.
//OPT14#7 JOB
//STEP1 EXEC PLIXCLG,PARM.PLI='SIZE(130K),OBJECT'
//PLI.SYSPN DD *
EX106: PROC OPTIONS(MAIN);

DCL RETURN_CODE FIXED BIN(31,0);
CALL PLISRTA (' SORT FIELDS=(7,74,CH,A) ':
  ' RECORD TYPE=F,LENGTH=(80) '
45000,
  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);
END EX106;

*/

//GO.SORTIN DD *
003329HOOKER S.W. RIVERDALE, SATCHEWELL LANE, BACONSFIELD
002886BOOKER R.R. Rotorua, Milkedge Lane, Tolley
003077ROOKER & SON, LITTLETON NURSERIES, SHOLTSPAR
059334HOOKE E.H. 109 ELMTREE ROAD, GANNET PARK, NORTHAMPTON
073872HOME TAVERN, WESTLEIGH
00931FOREST, IVER, BUCKS

//GO.SORTOUT DD SYSOUT=A,DCB=(RECFM=F,BLKSIZE=80)
//GO.SYSOUT DD SYSOUT=A
//GO.SORTLIB DD DSN=SYSL.SORTLIB,DISP=SHR
//GO.SORTWK01 DD UNIT=2314,SPACE=(TRK,20,CONTIG)
//GO.SORTWK02 DD UNIT=2314,SPACE=(TRK,20,CONTIG)
//GO.SORTWK03 DD UNIT=2314,SPACE=(TRK,20,CONTIG)

Figure 14-7. Invoking sort/merge via entry point PLISRTA

|USING USER EXIT E15 TO PASS RECORDS TO BE SORTED (PLISRTB) |

The example in Figure 14-8 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/merge program and tests the return code when processing is complete. Note that records to be sorted can be supplied only by the procedure invoked by way of user exit E15 (in this case, procedure E15X).

Each time procedure E15X is invoked by the sort/merge program, E15X 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 (PLISRTC) |

The example in Figure 14-9 illustrates the use of entry point PLISRTC to enable records to be supplied from the sort to the PL/I procedure E35X. As in previous examples, the main procedure invokes the sort/merge program and tests the return code when processing is complete. 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 14-10 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/merge program and tests the return code when processing is complete. The use of the E15 user exit is identical to that in Figure 14-8; the use of the E35 user exit is identical to that...
/* This procedure obtains records from the input stream */
PROC RETURNS(CHAR(80));
  DCL SYSIN FILE RECORD INPUT,
      INFIELD CHAR(80) INIT(' ');
  ON END FILE(SYSIN) BEGIN;
      PUT SKIP(3) EDIT ('END OF SORT PROGRAM INPUT')(A);
      CALL PLIRETC(8); /* SIGNAL END OF SORT INPUT */
      GOTO END E15;
  END;

  READ FILE (SYSIN) INTO (INFIELD);
  CALL PLIRETC(12); /* INPUT TO SORT CONTINUES */
  END E15;
END EX107;
*/

// GO. SYSIN DD *
003329HOKKER S.W. RIVERDALE, SATCHWELL LANE, BACONSFIELD
002886BOOKER R.R. ROTORUA, MILKEDGE LANE, TOBELEY
003077ROOKER & SON, LITTLETON NURSERIES, SHOLTSPAR
059334HOOK E.H. 109 ELMTREE ROAD, GANNET PARK, NORTHAMPTON
073872HOME TAVERN, WESTLEIGH
000931FOREST, IVER, BUCKS
// GO. SORTOUT DD SYSOUT=A, DCB=(RECFM=F, BLKSIZE=80)
// GO. SYSOUT DD SYSOUT=A
// GO. SORTLIB DD DSN=SYS1.SORTLIB, DISP=SHR
// GO.SORTWK01 DD UNIT=2314, SPACE=(TRK, 20, CONTIG)
// GO.SORTWK02 DD UNIT=2314, SPACE=(TRK, 20, CONTIG)
// GO.SORTWK03 DD UNIT=2314, SPACE=(TRK, 20, CONTIG)

Figure 14-8. Invoking sort/merge via entry point PLISRTB

The sequence of events is as 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 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.

SORTING VARIABLE-LENGTH RECORDS
The following points should be considered when sorting variable-length records:

- The portion of a variable-length record that contains the control field or fields on which the sort is to be performed must be present and of the same length for every record to be
//OPT14#9 JOB
//STEP1 EXEC PLIXCLG,PARM.PLI='SIZE(130K),OBJECT'
//PLI.SYSIN DD *
EX108:  PROCOPTIONS(MAIN);

   DCL RETURN_CODE FIXED BIN(31,0);
   CALL PLISRTC (' SORT FIELDS=(7,74,CH,A) ',
               ' RECORD TYPE=F,LENGTH=(80) ',
               45000,
               RETURN_CODE,
               E35X);
   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);

E35X:   /* THIS PROCEDURE OBTAINS SORTED RECORDS */
   PROC (INREC);
      DCL INREC CHAR(80);
      PUT SKIP EDIT (INREC)(A);
      CALL PLIRETC(4); /* REQUEST NEXT RECORD FROM SORT */
   END E35X;
   END EX108;

/*
//GO.SYSOUT DD SYSOUT=A
//GO.SORTLIB DD DSN=SYS1.SORTLIB,DISP=SHR
//GO.SORTWK01 DD UNIT=2314,SPACE=(TRK,20,,CONTIG)
//GO.SORTWK02 DD UNIT=2314,SPACE=(TRK,20,,CONTIG)
//GO.SORTWK03 DD UNIT=2314,SPACE=(TRK,20,,CONTIG)
//GO.SORTIN DD *
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 14-9. Invoking sort/merge via entry point PLISRTC

184  OS PL/I Optimizing Compiler: Programmer's Guide
EXEC PLIXCLG, PARM='PLI='SIZE(130K),OBJECT'  

DCL RETURN_CODE FIXED BIN(31,0);  

CALL PLISRTD (' SORT FIELDS=(7,74,CH,A) ',  
               ' RECORD TYPE=F,LENGTH=(80) ',  
               45000,  
               RETURN_CODE,  
               E15X,  
               E35X);  

IF RETURN_CODE = 16 THEN PUT SKIP ('SORT FAILED')(A);  
ELSE IF RETURN_CODE = 0 THEN PUT SKIP ('SORT COMPLETE')(A);  
ELSE PUT SKIP ('INVALID SORT RETURN CODE')(A);  

E15X: /* THIS PROCEDURE OBTAINS RECORDS FROM THE INPUT STREAM */  
  PROC OPTIONS CMAIN);  
    DCL RETURN_CODE FIXED BIN(31,0);  
    CALL PLISRTD (' SORT FIELDS=(7,74,CH,A) ',  
               ' RECORD TYPE=F,LENGTH=(80) ',  
               45000,  
               RETURN_CODE,  
               E15X,  
               E35X);  
    IF RETURN_CODE = 16 THEN PUT SKIP ('SORT FAILED')(A);  
    ELSE IF RETURN_CODE = 0 THEN PUT SKIP ('SORT COMPLETE')(A);  
    ELSE PUT SKIP ('INVALID SORT RETURN CODE')(A);  

E35X: /* THIS PROCEDURE OBTAINS SORTED RECORDS */  
  PROC (INREC);  
    DCL RETURN_CODE FIXED BIN(31,0);  
    CALL PLISRTD (' SORT FIELDS=(7,74,CH,A) ',  
               ' RECORD TYPE=F,LENGTH=(80) ',  
               45000,  
               RETURN_CODE,  
               E15X,  
               E35X);  
    IF RETURN_CODE = 16 THEN PUT SKIP ('SORT FAILED')(A);  
    ELSE IF RETURN_CODE = 0 THEN PUT SKIP ('SORT COMPLETE')(A);  
    ELSE PUT SKIP ('INVALID SORT RETURN CODE')(A);  

Figure 14-10. Invoking sort/merge via entry point PLISRTD
/* PL/I EXAMPLE USING PLISRTB TO SORT VARIABLE-LENGTH RECORDS */

EX1306: 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) ',
  45000
  RETURN_CODE,
  E15X);
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);
E15X: PROC RETURNS (CHAR(80) VARYING);
DCL STRING CHAR(80) VAR;
ON ENDFILE(SYSIN) BEGIN:
  PUT SKIP EDIT C'END OF INPUT') CAl;
  CALL PLIRETC(8);
  GOTO ENDE15;
END;
GET EDITCSTRING) (A(80»;
I=INDEXCSTRINGII', ' ')-1; /* RESET THE LENGTH OF THE */
STRING = SUBSTRCSTRING,l,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);
END15: END E15X;
END EX1036;

Figure 14-11. Sorting variable-length records

sorted. A sort cannot be performed on control fields whose length or position
within the record is liable to alter. Thus the control fields would be
expected within the minimum length given
for the records in the record statement.

- The length of each record is recorded in the
  first four bytes of the record.
  Provision for this length field should be made when you specify the sort
  control fields in the SORT control
  statement.
- Varying-length strings passed from an
  E15 procedure will have the PL/I length

field converted into a length-field for variable-length records. The length
will be the current length of the character string plus four bytes for the
length field. A fixed-length string will have the length field added to form
a variable-length record. Consequently, fixed-length strings of different
lengths can be returned from the same
procedure.
- An E15 procedure can only use adjustable strings to receive variable-length
  records from the sort program.

The example in Figure 14-11 shows a
program to read fixed-length (80-byte) records from SYSIN, convert them to varying-length strings, sort them, and print them. The maximum length of each variable-length record after conversion from a string is 84 bytes including the four-byte control field. The maximum block size for these records in the SORTOUT data set is 88 bytes. The sort field starts on byte 7 of the string. Consequently, it is defined as starting on byte 11 of the record to allow for the four-byte control field.
The PL/I Checkpoint/Restart feature provides a convenient method of taking checkpoints during the execution of a long-running program in a batch environment. It cannot be used in a TSO environment.

At points specified in the program, information about the current status of the program is written as a record on a data set. If the program terminates due to a system failure, this information can be used to restart the program close to the point where the failure occurred, avoiding the need to rerun the program completely.

This restart can be either automatic or deferred. An automatic restart is one that takes place immediately (provided the operator authorizes it when requested by a system message). A deferred restart is one that is performed later as a new job.

You can request an automatic restart from within your program without a system failure having occurred.

PL/I Checkpoint/Restart uses the Advanced Checkpoint/Restart Facility of the operating system. This is fully described in the publication Advanced Checkpoint/Restart.

To use checkpoint/restart you must do the following:

- Request, at suitable points in your program, that a checkpoint record is written. This is done with the built-in subroutine PLICKPT.

- Provide a data set on which the checkpoint record can be written.

- Also, to ensure the desired restart activity, you may need to specify the RD parameter in the EXEC or JOB statement (see the publication JCL Reference).

Note: You should be aware of the restrictions affecting data sets used by your program. These are detailed in the publication Advanced Checkpoint/Restart.

Writing a Checkpoint Record

Each time you want a checkpoint record to be written, you must invoke, from your PL/I program, the built-in subroutine PLICKPT.

The CALL statement has the form:

```
CALL PLICKPT((ddname[,check-id[,org[,code]]]));
```

The four arguments are all optional. If an argument is not used, it need not be specified unless another argument that follows it in the given order is specified. In this case, the unused argument must be specified as a null string. The following paragraphs describe the arguments.

"ddname" is a character string constant or variable specifying the name of the DD statement defining the data set that is to be used for checkpoint records. If this argument is omitted, the system will use the default ddname SYSCHK.

"check-id" is a character string constant or variable specifying the name that you want to assign to the checkpoint record so that you can identify it later, if required. If this argument is omitted, the system will supply a unique identification and print it at the operator's console.

"org" is a character string constant or variable specifying the attributes of the checkpoint data set. PS indicates sequential (that is, CONSECUTIVE) organization; PO represents partitioned organization. If this argument is omitted, PS is assumed.

"code" is a variable with the attributes FIXED BINARY (31), which can receive a return code from PLICKPT. The return code has the following values:

- 0 A checkpoint has been successfully taken.
- 4 A restart has been successfully made.
- 8 A checkpoint has not been taken. The PLICKPT statement should be checked.
- 12 A checkpoint has not been taken. Check for a missing DD statement, a hardware error, or insufficient space in the data set. A checkpoint will fail if taken while a DISPLAY statement with the REPLY option is still incomplete or if the program is using multitasking.
- 16 A checkpoint has been taken, but ENQ macro calls are outstanding and will not be restored on restart. This
situation will not normally arise for a PL/I program.

Checkpoint Data Set

A DD statement defining the data set on which the checkpoint records are to be placed, must be included in the job control procedure. This data set can have either CONSECUTIVE or partitioned organization. Any valid ddname can be used. If you use the ddname SYSCHK, you do not need to specify the ddname when invoking PLICKPT.

A data set name need be specified only if you want to keep the data set for a deferred restart. The I/O device can be any magnetic-tape or direct-access device.

If you want to obtain only the last checkpoint record, then specify status as NEW (or OLD if the data set already exists). This will cause each checkpoint record to overwrite the previous one.

If you want to retain more than one checkpoint record, specify status as MOD. This will cause each checkpoint record to be added after the previous one.

If the checkpoint data set is a library, then "check-id" is used as the member-name. Thus a checkpoint will delete any previously-taken checkpoint with the same name.

For direct access storage, enough primary space should be allocated to store as many checkpoint records as you will retain. You can specify an incremental space allocation, but it will not be used. A checkpoint record is approximately 5000 bytes longer than the area of main storage allocated to the step.

No DCB information is required, but you can include any of the following, where applicable:

OPTCD=W, OPTCD=C, RECFM=UT, NCP=2, TRTCH=C

These subparameters are described in Appendix A.

Performing a Restart

A restart can be automatic or deferred. Automatic restarts can be made after a system failure or from within the program itself. All automatic restarts need to be authorized by the operator when requested by the system.

AUTOMATIC RESTART AFTER A SYSTEM FAILURE

If a system failure occurs after a checkpoint has been taken, the automatic restart will occur at the last checkpoint if you have specified RD=R (or omitted the RD parameter) in the EXEC or JOB statement.

If a system failure occurs before any checkpoint has been taken, then an automatic restart, from the beginning of the job step, can still occur if you have specified RD=R in the EXEC or JOB statement.

If a system failure occurs after a checkpoint has been taken, then you can still force automatic restart from the beginning of the job step by specifying RD=RNC in the EXEC or JOB statement.

AUTOMATIC RESTART FROM WITHIN THE PROGRAM

An automatic restart can be requested at any point in your program. The rules applying to the restart are the same as for a restart after a system failure. To request the restart, you must execute the statement:

CALL PLIREST;

To effect the restart, the compiler terminates the program abnormally, with a system completion code of 4092. Therefore, to use this facility, the system completion code 4092 must not have been deleted from the table of eligible codes at system generation.

DEFERRED RESTART

To ensure that automatic restart activity is canceled, but that the checkpoints are still available for a deferred restart, specify RD=NR in the EXEC or JOB statement when the program is first executed.

If a deferred restart is subsequently required, the program must be submitted as a new job, with the RESTART parameter in the JOB statement. The RESTART parameter specifies the job step at which the restart is to be made and, if you want to restart at a checkpoint, the name of the checkpoint record. The RESTART parameter has the form:

RESTART=(stepname[,check-id])
For a restart from a checkpoint, you must also provide, immediately before the EXEC statement for the job step, a DD statement, with the name SYSCHK, defining the data set containing the checkpoint record.

MODIFYING CHECKPOINT/RESTART ACTIVITY

You can cancel automatic restart activity from any checkpoints taken in your program by executing the statement:

CALL PLICANC;

However, if you have specified RD=R or RD=RNC in the JOB or EXEC statement, automatic restart can still take place from the beginning of the job step.

Also, any checkpoints already taken will still be available for a deferred restart.

You can cancel any automatic restart, and also the taking of checkpoints, even if requested in your program, by specifying RD=NC in the JOB or EXEC statement.
Appendix A: DCB Subparameters

This appendix shows you how to code data set information in the DCB parameter of the DD statement and how to make use of existing DCB information. It also contains an alphabetic list of the subparameters that apply to a PL/I program. These subparameters are specified in the DCB parameter of the DD statement. Chapter 3 shows you how to write a DD statement and chapter 6 shows you how to use the name (ddname) of the DD statement. For a full description of the DD statement see the job control language publications.

DCB Parameter

The DCB parameter enables you to add information about your data set to the data control block (DCB) generated when the associated file in your PL/I program is opened. The information to be added is defined in one or more subparameters. These subparameters correspond to the operands of the DCB macro instruction and are specified in the same way. For a full description of macro instructions see the supervisor and data management macro instructions publication.

Code the DCB parameter as follows:

DCB=subparameter

or

DCB=(subparameter,subparameter)

For example:

DCB=BLKSIZE=80

DCB=(RECFM=FB,LRECL=80)

Using Existing DCB Information

You can use the DCB parameter to make use of DCB information that already exists either in the label of a similar data set, or that has been specified in the DCB parameter of an earlier DD statement.

INFORMATION IN SIMILAR DATA SETS

You can copy DCB information from the label of a similar data set by coding:

DCB=dsname

where "dsname" is the name of the data set containing the information you want to copy. This data set must be cataloged, it must be on a direct-access storage device, and the volume containing it must be mounted before execution of the job step.

INFORMATION IN AN EARLIER DATA SET

You can also copy the DCB information from an earlier DD statement in a job by coding:

DCB=* .stepname.ddname

where the asterisk tells the operating system that this is to be a backward reference, "stepname" is the name of the job step in which the earlier DD statement appears, and "ddname" is the name of the earlier DD statement. If the earlier DD statement is in a cataloged procedure you must include the procedure step name as well as the job step name, for example by coding:

DCB=* .stepname.procstepname.ddname

Overriding Existing DCB Information

If the existing DCB information does not meet your requirements exactly you can override any of the subparameters by specifying the required information in a new subparameter. For example, if an existing DD statement named IN in a job step named COMP has the following DCB parameter:

DCB=(RECFM=FB,LRECL=80)

and you want LRECL to be 100, simply code:

DCB=(RECFM=FB,LRECL=100)
Subparameters of the DCB Parameter

The following is a summary of those subparameters that can apply to your PL/I program. The notation used in the descriptions is as follows:

- \( n \) unsigned decimal integer
- \{ \} indicates a choice
- \( [] \) brackets indicate that the item enclosed is optional.

Code capital letters and numbers exactly as shown.

**BLKSIZE=n**

specifies the length in bytes of a block. The maximum length is 32760 bytes.

For fixed-length records, the block size must be an integral multiple of the record length (LRECL); the minimum size is 1 byte.

For variable-length (V-format and VB-format) records, the block size must be at least eight bytes larger than the largest item of data that you expect to read or write (that is, four bytes larger than the record length specified in LRECL). However, if the records are spanned (VS-format and VBS-format), you can specify block size independently of record length. The minimum block size for data sets on magnetic tape is 18 bytes.

**BUFNO=n**

specifies the number of buffers to be used in accessing the data set. The maximum number is 255 (unless another maximum has been determined for your installation during system generation). For a STREAM file or BUFFERED RECORD file, if you do not specify the number of buffers or you specify zero buffers, the number is assumed to be two.

**CODE=A|B|C|F|I|N|T**

specifies the code in which paper tape is punched. (Data is read into main storage and then converted from that code to EBCDIC.)

- A ASCII (8-track)
- B Burroughs (7-track)
- C NCR (8-track)
- F Friden (8-track)
- I IBM BCD perforated-tape transmission code (8-track)
- N No conversion required (F-format records only)
- T Teletype (5-track)

If no code is specified, I is assumed.

**CYLOFL=n**

specifies, for an INDEXED data set, the number of tracks of each cylinder to be reserved for the records that overflow from other tracks in that cylinder. The theoretical maximum is 99, but the practical limit varies with the particular device.

There must be at least one track in each cylinder to hold the prime data.

**DEN=0|1|2|3**

specifies the recording density for magnetic tape as follows:

<table>
<thead>
<tr>
<th>Bytes per inch (bpi)</th>
<th>7-track</th>
<th>9-track</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>200</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>556</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>800</td>
<td>800</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>1600</td>
</tr>
</tbody>
</table>

The density assumed if you omit this subparameter is:

- 7-track: 1600 bpi
- 9-track (single density): 800 bpi
- 9-track (dual density): 1600 bpi

(The subparameter TRTCB is required for 7-track tape.)
DSORG=IS|DA

specifies the organization of the data set you are creating:
IS (indexed sequential): INDEXED data set
DA (direct access): REGIONAL data set

This subparameter is not required for CONSECUTIVE data sets.

FUNC=function

specifies the function or functions to be performed by an IBM 3525. The following values of "function" are valid:

- I - Interpret
- R - Read
- P - Punch
- W - Print
- D - Data protection
- X - Printer
- T - Two-line printer

The letters have the following meanings:

- I - Interpret
- R - Read
- P - Punch
- W - Print
- D - Data protection
- X - Printer
- T - Two-line printer

KEYLEN=n

specifies the length in bytes of the recorded key of records in INDEXED, REGIONAL(2), and REGIONAL(3) data sets. The maximum key length is 255 bytes.

LIMCT=n

limits the extent of the search for a record or space to add a record in a REGIONAL(2) or REGIONAL(3) data set beyond the region number specified in the source key.

If you do not specify a limit, the search starts at the specified region and continues through the whole of the data set.

For REGIONAL(2), LIMCT specifies the number of records to be searched. The search starts at the beginning of the track on which the record is situated and continues to the end of the track containing the last record to be searched.

For REGIONAL(3), LIMCT specifies the number of tracks to be searched.

LRECL=n

specifies the length of a record in bytes; the maximum length is 32760 bytes for F-format records, and 32756 bytes for V-format records. You must specify a record length for blocked records.

For F-format and FB-format records, the record length must not exceed the block size (BINSIZE) value; the minimum length is 1 byte.

For V-format records, give the maximum record length including the four control bytes required by the operating system; the minimum record length for V-format records is 14 bytes (ten bytes of data and four control bytes). The record length for V-format and VS-format records must be at least four bytes less than the block size (BLSIZE) value; however, for VS-format and VBS-format records, it can be specified independently of block size.

MODE={E|C}[O|R]

specifies the mode of operation for a card reader or punch. E indicates EBCDIC, and C indicates column binary. O specifies Optical Mark Read mode on an IBM 3505, and R specifies Read Column Eliminate mode on an IBM 3505 or 3525.

NCP=n

specifies the number of channel programs allocated to a file when it is opened; the number of simultaneous input/output operations on the file (that is, the number of incomplete event variables) cannot exceed the number of channel programs. The NCP subparameter applies only to direct access to INDEXED data sets, or sequential access to CONSECUTIVE or REGIONAL data sets that are unbuffered. The maximum number of channel programs is 99 (unless another maximum was established for your installation at system generation); the default value assumed if you omit the subparameter is 1.
For DIRECT access to an INDEXED data set, simultaneous input/output operations in excess of the number of channel programs are queued until a channel program becomes available.

For UNBUFFERED SEQUENTIAL access to CONSECUTIVE or REGIONAL data sets, the ERROR condition is raised if there are too many concurrent operations.

The NCP subparameter overrides the BUFNO subparameter or the BUFFERS option of the ENVIRONMENT attribute. One buffer is allocated for each channel program.

\[ NTM=n \]

specifies, for an INDEXED data set, the number of tracks in the cylinder index referred to by each master index entry, and the number of tracks within each level of the master index referred to by each entry in the next higher level. The maximum value for \( n \) is 99. (See also OPTCD=M later in this chapter.)

\[ OPTCD=\text{option list} \]

lists optional data management services. To indicate the services you require, code the appropriate letters (see below) without separating blanks, in place of "option list" (for example, OPTCD=LY).

OPTCD=C requests chained scheduling, which improves input/output performance by reducing the time required to transmit blocks to and from auxiliary storage devices. In chained scheduling, the data management routines bypass the normal input/output scheduling routines and chain several input/output operations together; a series of read operations, for example, is issued as a single chain of commands instead of several separate commands.

Chained scheduling is most useful in programs whose performance is input/output limited. If you use this feature, you should request at least three buffers or at least three channel programs. Chained scheduling can be used with CONSECUTIVE or REGIONAL SEQUENTIAL data sets; it should not be used for INPUT or UPDATE with U-format records.

OPTCD=U requests a write validity check for a direct-access device.

OPTCD=Y requests that the data management routines use the cylinder overflow area for overflow records in an INDEXED data set. The size of the overflow area is established by CYLOFL=n.

\[ \text{RECFM=} \]

indicates the record format as follows:

\[ \{ \text{F[\text{B}]} \ [\text{S}] \} \]

\[ \{ \text{V[\text{B}]} \ [\text{S}] \} \ [\text{T}] \ [\text{A|M}] \]

\[ \text{U} \]

lists optional data management services. To indicate the services you require, code the appropriate letters (see below) without separating blanks, in place of "option list" (for example, OPTCD=LY).

OPTCD=C requests chained scheduling, which improves input/output performance by reducing the time required to transmit blocks to and from auxiliary storage devices. In chained scheduling, the data management routines bypass the normal input/output scheduling routines and chain several input/output operations together; a series of read operations, for example, is issued as a single chain of commands instead of several separate commands.

Chained scheduling is most useful in programs whose performance is input/output limited. If you use this feature, you should request at least three buffers or at least three channel programs. Chained scheduling can be used with CONSECUTIVE or REGIONAL SEQUENTIAL data sets; it should not be used for INPUT or UPDATE with U-format records.

OPTCD=U requests a write validity check for a direct-access device.

OPTCD=Y requests that the data management routines use the cylinder overflow area for overflow records in an INDEXED data set. The size of the overflow area is established by CYLOFL=n.

\[ \text{RECFM=} \]

indicates the record format as follows:

\[ \{ \text{F[\text{B}]} \ [\text{S}] \} \]

\[ \{ \text{V[\text{B}]} \ [\text{S}] \} \ [\text{T}] \ [\text{A|M}] \]

\[ \text{U} \]

OPTCD=\text{option list}

lists optional data management services. To indicate the services you require, code the appropriate letters (see below) without separating blanks, in place of "option list" (for example, OPTCD=LY).

OPTCD=C requests chained scheduling, which improves input/output performance by reducing the time required to transmit blocks to and from auxiliary storage devices. In chained scheduling, the data management routines bypass the normal input/output scheduling routines and chain several input/output operations together; a series of read operations, for example, is issued as a single chain of commands instead of several separate commands.

Chained scheduling is most useful in programs whose performance is input/output limited. If you use this feature, you should request at least three buffers or at least three channel programs. Chained scheduling can be used with CONSECUTIVE or REGIONAL SEQUENTIAL data sets; it should not be used for INPUT or UPDATE with U-format records.

OPTCD=U requests a write validity check for a direct-access device.

OPTCD=Y requests that the data management routines use the cylinder overflow area for overflow records in an INDEXED data set. The size of the overflow area is established by CYLOFL=n.

\[ \text{RECFM=} \]

indicates the record format as follows:

\[ \{ \text{F[\text{B}]} \ [\text{S}] \} \]

\[ \{ \text{V[\text{B}]} \ [\text{S}] \} \ [\text{T}] \ [\text{A|M}] \]

\[ \text{U} \]

OPTCD=\text{option list}

lists optional data management services. To indicate the services you require, code the appropriate letters (see below) without separating blanks, in place of "option list" (for example, OPTCD=LY).

OPTCD=C requests chained scheduling, which improves input/output performance by reducing the time required to transmit blocks to and from auxiliary storage devices. In chained scheduling, the data management routines bypass the normal input/output scheduling routines and chain several input/output operations together; a series of read operations, for example, is issued as a single chain of commands instead of several separate commands.

Chained scheduling is most useful in programs whose performance is input/output limited. If you use this feature, you should request at least three buffers or at least three channel programs. Chained scheduling can be used with CONSECUTIVE or REGIONAL SEQUENTIAL data sets; it should not be used for INPUT or UPDATE with U-format records.

OPTCD=U requests a write validity check for a direct-access device.

OPTCD=Y requests that the data management routines use the cylinder overflow area for overflow records in an INDEXED data set. The size of the overflow area is established by CYLOFL=n.

\[ \text{RECFM=} \]

indicates the record format as follows:

\[ \{ \text{F[\text{B}]} \ [\text{S}] \} \]

\[ \{ \text{V[\text{B}]} \ [\text{S}] \} \ [\text{T}] \ [\text{A|M}] \]

\[ \text{U} \]
operating system feature that can be incorporated during system generation. It allows a block to overflow from one track of a direct-access device to another. Track overflow is useful in achieving greater data-packing efficiency, and also allows the size of a record to exceed the capacity of a track.

### Notes

**Data conversion and translation:** Data on 9-track magnetic tape, like that in main storage, is held in 8-bit bytes, a ninth bit being used for parity checking; data on 7-track tape is held in the form of 6-bit characters with a parity bit. The conversion feature of the 2400 series magnetic-tape drives treats all data as if it were in the form of a bit string, breaking the string into groups of six bits for writing on 7-track tape, or into groups of eight bits for reading into main storage. The translation feature changes the form in which character data is held from 8-bit EBCDIC to 6-bit ECD or vice versa. If you specify neither conversion nor translation, only the last six bits of each 8-bit byte are transmitted; the first two are lost on output and are set to zero on input.

**Parity:** Odd parity checking is normally used in IBM System/360, but you should specify even parity if you want to read a tape that was written by a system using even parity, or to write a tape for a system that demands even parity.

### Choice of technique

The use of a technique other than C restricts the character set in which data can be written if it is subsequently to be reread and result in the same bit configuration in main storage. (An 8-bit code offers 256 possible configurations, but a 6-bit code only 64.) For stream-oriented or record-oriented transmission of character strings or pictured data, you can use technique C or T; you can also specify ET if your program is written in the 48-character set. (Seven-track tape recording systems indicate a zero bit by the absence of
magnetization of the tape. Even parity
checking does not allow the code 000000 to
be used to represent the character zero,
since an unmagnetized band is not
acceptable on the tape. Therefore the code
that would otherwise represent a colon (:) is used for the character zero, precluding
the use of the full PL/I 60-character set.)
For record-oriented transmission of
arithmetic data, you must specify technique C.

Block Size: A PL/I program cannot be used
to access a data set on a 7-track tape
recorded with a block size that is not a
multiple of four bits. For example, a 7-
track tape with a block size of 175 has
6*175 bits, that is, 1,050 bits in each
block and cannot be read unless it is first
modified by the the IEBCOPY utility
program.
Appendix B: Compatibility with the PL/I(F) Compiler

Some features of the PL/I Optimizing Compiler implementation are incompatible with the PL/I (F) Compiler implementation. The most significant incompatibilities are listed below. In every case, the description given is of the optimizing compiler implementation. Programs which were written for the (F) compiler and which 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 of this compiler. The language reference manual for this compiler gives full details of the implementation of each language feature.

Areas

The (F) compiler holds the length of an area in the first 16 bytes of the area. The optimizing compiler holds the length of an area in a descriptor outside the area.

Arrays and Structures

- The maximum number of dimensions in an array is 15.

- The maximum depth of a structure is 15.

Built-in Functions

- Built-in functions are recognized on the basis of context only, so that all programmer-defined external procedures must be declared explicitly. Built-in functions and pseudovariables without arguments, such as TIME and ONCHAR, must also be declared explicitly with the BUILTIN attribute, or contextually with a null argument list, for example: TIME( ).

- For a variable to be a valid argument to the ADDR built-in function it must be connected and its left extremity must not lie within bytes that contain data belonging to other variables.

- The ALLOCATION built-in function returns a fixed-binary value giving the number of generations of the argument that exist in the current task.

- The NULLO built-in function is not implemented in the optimizing compiler. The NULL built-in function can be used for offset variables as well as for pointer variables.

- The ONCOUNT built-in function can be used in any on-unit and gives the number of interrupts remaining to be processed at any stage in the execution of the current task. In particular, this includes event and non-event I/O and multiple computational interrupts. In the case of event I/O, the value of ONCOUNT is the number of remaining exceptional conditions to be processed as a result of the execution of the WAIT statement.

- When using REGIONAL(1) organization, the value returned by the ONKEY built-in function for a specification error consists of the last eight bytes of the source key, padded on the right with blanks if necessary. This value is returned for all I/O conditions other than ENDFILE, or other than ERROR raised as standard system action for an I/O condition.

- In a RECORD I/O statement with the KEY or KEYFROM option, the ONKEY built-in function returns a null string when the ERROR condition is raised.

- In a RECORD I/O statement referring to a KEYED file (but with no KEY, KEYFROM, or KEYTC option specified) the ONKEY built-in function returns the recorded key.

- The PROD built-in function accepts arguments that are arrays of either fixed-point or floating-point elements. The value returned has the same scale as the argument given, except for fractional fixed-point arguments for which the result is in floating-point.

- If the first argument of the ROUND built-in function is a string, it is converted to arithmetic and rounded; the first argument must be convertible to arithmetic. Also, a different formula is used to determine the precision of a
fixed-point result.
- The SUBSTR built-in function returns a non-varying string.
- The SUM built-in function accepts arguments that are arrays of either fixed-point or floating-point elements. The value returned has the same scale as the argument given.
- The arguments of the TRANSLATE built-in function are converted to character-strings in all cases.
- The first 16 bits of the result returned by the UNSPEC built-in function for a varying string argument represent the current length of the string.

Checkpoint/R Restart

The arguments to the function PLICKPT are mandatory.

Conditions

- When used with arrays, the CHECK condition is raised after assignment to each element. The standard system action when an assignment is made to a single element of an array is to print the value of only the element assigned.
- The STRING RANGE condition is not raised for SUBSTR(string, i, 0) when "i" is one greater than the length of "string". A null string is returned.

Control Variable in DO statement

The pseudovariables COMPLETION, COMPLEX, PRIORITY, and STRING are not allowed as the control variable of a DO statement.

DEFINED Attribute

- Simple defining of strings and areas on a larger base is allowed.

For example:

```
DECLARE A (6) CHAR (6),
     B (3) CHAR (3) DEFINED A;
```

This example will result in simple defining - B(1) will refer to the first three characters of A(1), B(2) to the first three characters of A(2), and so on.

If string overlay defining is required, the user must specify POSITION (1) on the declaration of the defined item (B in the example above).

If the string lengths or bounds of the defined item cannot be contained in the base, simple defining will be assumed.

For example:

```
DECLARE A (6) CHAR (6),
     B (7) CHAR (3) DEFINED (A);
```

In this example, simple defining will be used because the bounds of array B exceed the bounds of array A.

- If the DEFINED attribute is used with an array of pictures, the defined item must match the base item exactly.

Dependent Declarations

Only one level of dependent declaration is allowed.

DISPLAY Statement

The maximum length of the reply is 72 characters.

Dumps from PL/I Programs

The object-time dump facility of the (F) compiler, IHEDUMP, requires a DD statement with the ddbname PLIDUMP. Its equivalent in the optimizing compiler, PLIDUMP, requires a DD statement with the ddbname PLIDUMP. The optimizing compiler will attempt to use PLIDUMP if PLIDUMP is not available.

ENDPAGE condition

When ENDPAGE is signaled it cannot be raised again on the same page, except by the use of a further SIGNAL statement.
### Appendix B: Compatibility With the PL/I (F) Compiler

#### Old form

<table>
<thead>
<tr>
<th>Old form</th>
<th>Converted to</th>
</tr>
</thead>
<tbody>
<tr>
<td>F(b)</td>
<td>F BLKSIZE(b) RECSIZE(b)</td>
</tr>
<tr>
<td>F(b,r)</td>
<td>FB BLKSIZE(b) RECSIZE(r)</td>
</tr>
<tr>
<td>U(b)</td>
<td>U BLKSIZE(b) RECSIZE(b)</td>
</tr>
<tr>
<td>V(b)</td>
<td>V BLKSIZE(b) RECSIZE(b-4)</td>
</tr>
<tr>
<td>V(b,r)</td>
<td>VB BLKSIZE(b) RECSIZE(r)</td>
</tr>
<tr>
<td>VBS(b,r)</td>
<td>VBS BLKSIZE(b) RECSIZE(r)</td>
</tr>
<tr>
<td>VS(b,r)</td>
<td>VS BLKSIZE(b) RECSIZE(r)</td>
</tr>
</tbody>
</table>

Figure B-1. Environment options recognized by the compiler

---

**Entry Names, Parameters, and Returned Values**

- Each alternative entry expression in a GENERIC attribute is followed by a WHEN clause. The appearance of an entry name alternative does not constitute a declaration of the entry name. The alternative selected is the first for which each descriptor is a subset of the attributes of the corresponding argument in the generic reference.

- The dimension attribute is not allowed in a generic descriptor.

- In general, an entry name in parentheses causes a dummy variable to be created; for the function to be invoked, a null argument list is required. However, an entry name argument in parentheses, or an entry name without arguments, will be invoked if passed to a procedure whose parameter descriptor for the corresponding argument specifies an attribute other than ENTRY.

- External entry names must always be explicitly declared.

- Area and string extents in the RETURNS attribute or option must be represented by a decimal integer constant.

- The maximum depth of nesting in a descriptor list in the ENTRY attribute is 2.

- An aggregate expression involving strings may not be passed as an argument unless there is a corresponding parameter descriptor in which all string lengths are specified as decimal integer constants.

- An internal entry constant cannot be declared in a DECLARE statement. REDUCIBLE and IRREDUCIBLE may be specified on PROCEDURE and ENTRY statements. A scalar cannot be passed to an array parameter of an internal entry constant if the parameter's bounds are specified by asterisks.

---

#### Converted to

<table>
<thead>
<tr>
<th>Old form</th>
<th>Converted to</th>
</tr>
</thead>
<tbody>
<tr>
<td>G(m)</td>
<td>V TP(M) RECSIZE(m)</td>
</tr>
<tr>
<td>R(r)</td>
<td>V TP(R) RECSIZE(r)</td>
</tr>
</tbody>
</table>

Figure B-2. Teleprocessing environment options

---

**Error Correction**

The error correction logic differs from that used by the PL/I (F) compiler. Invalid programs that are compiled and corrected by the (F) compiler may not give the same results on the optimizing compiler.

---

**EXCLUSIVE Attribute**

- The EXCLUSIVE attribute implies only the RECORD attribute; DIRECT, UPDATE, and KEYED will apply only by default.

- The EXCLUSIVE attribute can be used in non-tasking programs, and jobs in the system can affect each other.
Facility | Entry-point Name
---|---
Sort | PLISRTA
| PLISRTB
| PLISRTC
| PLISRTD
Checkpoint/Restart | PLICKPT
| PLIREST
| PLICANC
| PLIRETC
Return Code | PLIDUMP

Figure B-3. Operating system facilities

Expression Evaluation

In a concatenation operation, a BINARY operand is converted to BIT if the other operand is BINARY or BIT.

FIXED BINARY Expressions

The length of FIXED BINARY constants and intermediate results with a precision less than 16 is 2 bytes. The UNSPEC built-in function returns a result whose length is 16 bits.

INITIAL Attribute

The limitations on the length of DECLARE statements imposes some restrictions on the use of the INITIAL attribute. These restrictions are described under "Statement Length" later in this appendix.

LIKE Attribute

The optimizing compiler does not permit a substructure to have the LIKE attribute when another substructure within the major structure is the object of a further LIKE attribute. For example:

DCL 1 A LIKE C,
1 B,
2 C,
3 D,
3 E,
2 F LIKE X,
1 X,
2 Y,
2 Z;

In this example, the structure A has the LIKE attribute which refers to substructure C in structure B. B also contains substructure F with the LIKE attribute.

Link-editing

Programs translated by the optimizing compiler cannot be link-edited with object modules produced by the PL/I (F) compiler.

Locked Records

- The locking action takes place at the data set level.
- If an on-unit is entered as a result of a REWRITE or DELETE statement, the record is unlocked if the on-unit is terminated by a GOTO statement as well as normal completion.
- The ERROR condition is raised if a file is closed while subtasks currently have records in it locked.
- A record is not locked due to "key not found" or "key outside data" conditions.

Multitasking Programs

In certain circumstances, it is possible for multitasking programs compiled by the PL/I Optimizing Compiler to take longer to execute than corresponding programs compiled by the PL/I (F) Compiler. This situation can occur if a subtask, in which no files are opened, is repeatedly called.

NAME Option

When using batched compilation to produce two object modules that will be combined into one load module, the NAME option should not be used in the EXEC statement, because the options in the PARM parameter apply to all procedures in the batch.

Operating System Facilities

- The operating system facilities for sorting, for checkpoint/restart, for generating a return code, and for
obtaining a dump are all invoked by means of a CALL statement with the appropriate entry-point name; for example CALL PLISRTA. The entry point names, which are listed in Figure B-3, have the BUILTIN attribute and need not be declared explicitly.

- The optimizing compiler does not recognize the entry names used by the PL/I (F) compiler, that is, IHESRTx, IHESARC, IHEDUMx, and IHECKPT. Existing programs for the PL/I (F) compiler that use these entry names must be amended so that the DECLARE statements for them are removed completely.

---

**Pictures**

Sterling picture data is not implemented. Therefore the following picture characters are not allowed:

- G, H, M, P, 6, 7, 8.

**Preprocessor**

- Text replaced by preprocessor statements does not have blanks appended to either end of the replacement value.

- A parameter descriptor list is not allowed in the declaration of a preprocessor variable with the ENTRY attribute.

- The RETURNS attribute may not be specified in a preprocessor DECLARE statement.

**Pseudovariables**

For a varying string, the first 16 bits of the value of the UNSPEC pseudovariable represent the current length of the string.

**Record I/O**

- If READ...KEY is used with a sequential data set and no record with the specified key exists in the data set, the KEY condition is raised and the file is positioned at the next record in ascending sequence.

- If an embedded key in a record is not identical to that specified in a WRITE...KEYFROM, LOCATE...KEYFROM, or REWRITE FROM...KEY statement, the latter is moved into the record.

- READ, REWRITE, and DELETE statements are invalid for REGIONAL DIRECT OUTPUT files.

- There is no default record format for RECORD files (unless the file is associated with a DD DUMMY statement, in which case the default is U-format). If the record format is not specified, the UNDEFINEDFILE condition is raised.

---

**Redundant Expression Elimination**

The optimization process of eliminating redundant expressions could give rise to an incompatibility for (F) compiler programs that are recompiled by the optimizing compiler. If a program contains an expression, such as IF (A=0) THEN (C=D) then... such that the condition (A=0) is satisfied, the expression (C=D) is ignored. However, (C=D) might contain a function which, if not evaluated, could give rise to error.

**Return Codes**

The maximum return code that may be set using PLIRETC is 999.

**REWIND Option**

The optimizing compiler does not implement the REWIND option. Programs that use this option should be modified. Note that the function of the REREAD option implemented by the optimizing compiler is different to that of the REWIND option. The main difference is that the REREAD option overrides any DISP parameter and controls the positioning of the magnetic tape volume whereas the REWIND option specifies that the DISP parameter is to control the positioning of the magnetic tape volume according to the subparameters specified.

**Standard File SYSPRINT**

The default record characteristics are: record format = VBA, record length = 125, and block size = 129. The default number of buffers is two. Using the DCB parameter

Appendix B: Compatibility With the PL/I (F) Compiler 203
of the DD statement, record format, record length, block size, and the number of buffers can be changed to any valid value.

The following sequence:

DCL SYSPRINT FILE;
   ...
OPEN FILE(SYSPRINT);

will cause the UNDEFINEDFILE condition to be raised. Omission of either or both statements will result in correct execution. The file should be declared implicitly or with the attributes STREAM and OUTPUT.

Statements

The approximate maximum number of statements in a program is 10,000.

Statement Labels

A label on a DECLARE statement is treated as if it were on a null statement.

Statement Lengths

The optimizing compiler has a restriction that any statement must fit into the compiler's work area. The maximum size of this work area varies with the amount of space available to the compiler. The limitations on the length of statements are given in Figure B-4.

<table>
<thead>
<tr>
<th>Space Available</th>
<th>Maximum Statement Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>50K - 55K</td>
<td>1012 characters</td>
</tr>
<tr>
<td>55K - 69K</td>
<td>1600 characters</td>
</tr>
<tr>
<td>Over 69K</td>
<td>3400 characters</td>
</tr>
</tbody>
</table>

Figure B-4. Statement length limitations

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

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

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

The (F) compiler can use up to 90K bytes for its work area. It is possible that programs with large DECLARE statements will not compile successfully on the optimizing compiler although they had compiled successfully on the (F) compiler. The following techniques are suggested to overcome this problem:

- Increase the main storage available to the compiler, unless it already exceeds 69K bytes.
- Simplify the DECLARE statement so that the compiler can treat the statement in the manner described above.
- Modify any lists of items following the INITIAL attribute so that individual items are smaller and separated by commas not contained in parentheses. For example, the following declaration is followed by an expanded form of the same declaration. The compiler can more readily accommodate the second declaration in its work area:

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

Stream Transmission

- A filemark (or end of string mark for the STRING option) is a valid item delimiter for list-directed and data-directed input. An item thus delimited is processed intact, and ENDFILE (ERROR for string) is raised on attempting to read a further item from the file or string.
- After processing a GET LIST statement, a file is positioned to the next non-blank
character or, if this is a comma, to the character following the comma. A GET EDIT statement following a GET LIST on the same file must take into account the position of the file.

- The NAME condition (ERROR for the STRING option) is raised for all errors (including out of range subscripts) detected on the left-hand side of an item of data-directed input.

- The COPY option causes all items to be copied, including any skipped by the SKIP option. A contextual declaration of SYSPRINT is caused if no file name is specified for the COPY option.

- When transmitting DATA-directed output to a PRINT file, data items less than or equal to the line size will not be split between lines. Data items greater than the line size will, if possible, be split between the equals sign and the data value.

- Execution of a PUT statement in which the LINE option specifies the current line causes the ENDPAGE condition to be raised unless the current column is 1.

**Varying-length Strings**

The (F) compiler initializes a varying-length string to a null string, the equivalent of:

```
STR = '';
```

whenever such a string is allocated. The optimizing compiler does not perform any initialization of varying-length strings, unless the INITIAL attribute is used.

**WAIT Statement**

If a WAIT statement requires the completion of an inactive and incomplete event variable in a non-tasking program, then after any I/O event variables named in the same statement are completed, a message is printed and the program is terminated. In a TSO environment, control passes to the terminal.
Appendix C: Requirements 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, he will first determine whether or not the malfunction really is a problem in the program product. If he decides that the program product is at fault, he must then check to see if the fault is a known fault for which he can obtain an existing fix-up. If the fault is not known, he must refer the problem 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 C-1 is supplied and if the source program is reduced to the smallest, least complex form that 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 programming 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 be put into abeyance until they are supplied.

**Job Control Statements**

Listings of job control statements used to run the program must be supplied. For OS installations, any local cataloged procedures should be shown in expanded form, obtained by specifying MLEVEL=1 in the JOB statement. 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 the IBM program maintenance personnel to reproduce the problem.

**Terminal Session Listing**

If a malfunction is detected during a conversational (TSO), session the entire
<table>
<thead>
<tr>
<th>Material Required</th>
<th>Compiler Option</th>
<th>When Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original source program</td>
<td>C,E,T</td>
<td></td>
</tr>
<tr>
<td>Job Control Statements</td>
<td>C,E,T</td>
<td></td>
</tr>
<tr>
<td>Operating Instructions/Console Log</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>Terminal session listing</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>LOGON procedure listing</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>Listings: Source listing</td>
<td>SOURCE (S)</td>
<td>C,E,T</td>
</tr>
<tr>
<td>Cross-reference listing</td>
<td>XREF (X)</td>
<td>C,E,T</td>
</tr>
<tr>
<td>Attribute table</td>
<td>ATTRIBUTES (A)</td>
<td>C,E,T</td>
</tr>
<tr>
<td>Aggregate table</td>
<td>AGGREGATE (AG)</td>
<td>C,E,T</td>
</tr>
<tr>
<td>Storage table</td>
<td>STORAGE (STG)</td>
<td>C,E,T</td>
</tr>
<tr>
<td>Compiler options</td>
<td>OPTIONS (OP)</td>
<td>C,E,T</td>
</tr>
<tr>
<td>Compiler terminal dump</td>
<td>DUMP (DU)</td>
<td>C,T</td>
</tr>
<tr>
<td>Linkage editor map</td>
<td>MAP (linkage editor option)</td>
<td>E,T</td>
</tr>
<tr>
<td>Execution-time dump</td>
<td>E,T</td>
<td></td>
</tr>
<tr>
<td>User subroutines</td>
<td>E,T</td>
<td></td>
</tr>
<tr>
<td>User data sets</td>
<td>E,T</td>
<td></td>
</tr>
<tr>
<td>Preprocessor input listing</td>
<td>INSOURCE (IS)</td>
<td>P,T</td>
</tr>
<tr>
<td>Preprocessor output</td>
<td>MDECK (MD)</td>
<td>C,E,T</td>
</tr>
<tr>
<td>List of applied PTFs</td>
<td></td>
<td>C,E,T</td>
</tr>
</tbody>
</table>

Note: "C" indicates the requirements for a compile-time error; "E" indicates requirements for an execution-time error; "P" indicates the requirements for a preprocessor error; "T" indicates the requirements for conversational (TSO) error.

Figure C-1. Summary of requirements for APAR submission

listing produced at the terminal, from LOGON to LOGOFF, must be supplied. If the failure occurs during conversational processing of a large program under TSO, it may be easier to obtain the program's output by listing the PRINT data set on a line printer.

LOGON Procedure

If a malfunction is detected during a conversational (TSO) session, a listing of the LOGON procedure must be supplied to enable the session to be reproduced.

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.

If the failure occurs during conversational compilation of a large program under TSO, it may be easier to obtain the required listings by running the compilation in a background region. Alternatively, the output PRINT data set may be printed on a line printer.

**Linkage Editor Map**

When a problem occurs at execution time, a linkage editor map that was obtained when the copy of the program that has failed was link edited 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 or by using the system SYSABEND or SYSDUMP facilities. 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 including the following statement in an ERROR on-unit that will be entered when the program fails:

```
CALL PLIDUMP('TFHB');
```

**Applied PTFs**

A list of any program temporary fixes (PTF) and local fixes (s/zaps) applied to either the compiler or its libraries must be supplied. The IBM service aid program IMAPTFLS, described in the publication *OS Service Aids*, Order No. GC28-6719, can be used to obtain from a program library a listing showing those members of the library that have PTF or local fixes, provided that when the fix was made, the correct system status index (SSI) was included in the library directory. However, if a module contains more than one temporary fix, only the last fix to be applied will be listed by the IMAPTFLS program.

**Submitting the APAR**

When submitting material for an APAR to IBM, ensure that any magnetic tapes and decks of punched 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:

- 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.

Appendix C: Requirements for Problem Determination and APAR Submission 209
This appendix explains how exceptions and interrupts in Models 91 and 195 are handled by the operating system. An exception is a hardware occurrence (such as an overflow error) which can cause a program interrupt. An interrupt is a suspension of normal program activities. There are many possible causes of interrupts, but the following discussion is concerned only with interrupts resulting from hardware exceptions.

IBM System/360 Models 91 and 195 are high-speed processing systems in which more than one instruction may be executed concurrently. As a result, an exception may be detected and an interrupt occur when the address of the instruction which caused the exception is no longer held in the central processing unit. Consequently, the instruction causing the interrupt cannot be precisely identified. Interrupts of this type are termed imprecise. When an exception occurs, the machine stops decoding further instructions and ensures that all instructions which were decoded prior to the exception are executed before honoring the exception. Execution of the remaining decoded instructions may result in further exceptions occurring. An imprecise interrupt in which more than one exception has occurred is known as a multiple-exception imprecise interrupt.

The optimizing compiler permits processing of imprecise interrupts only when the compiler option IMPRECISE is in effect. This is useful when debugging a program. The effect of the option is:

1. To cause the compiler to insert special "no-operation" instructions at certain points in the program to localize imprecise interrupts to a particular segment of the program, thus ensuring that interrupt processing results in the action specified in the source program. These "no-operation" instructions are generated:
   - Before an ON-statement.
   - Before a REVERT statement.
   - Before internal code to set the SIZE condition.
   - Before internal code to change prefix options.
   - Between statements if GONUMBER or GONUMBER apply.
   - For a null statement. (This feature provides the programmer with source language control over the timing of program interrupts.)

2. To provide facilities for:
   - Detecting multiple-exception imprecise interrupts.
   - Setting the value that is required by the ONCOUNT built-in function.
   - Raising the appropriate PL/I conditions.

The order of processing the exceptions is as follows:

1. PL/I conditions in the order:
   UNDERFLOW
   FIXEDOVERFLOW or SIZE
   ERROR (if system action is required for either FIXEDOVERFLOW or SIZE)
   FINISH (if system action is required for the previous ERROR condition)
   OVERFLOW
   ERROR (if system action is required for OVERFLOW)
   FINISH (if system action is required for the previous ERROR condition)
   ZERODIVIDE
   ERROR (if system action is required for ZERODIVIDE)
   FINISH (if system action is required for the previous ERROR condition)

Note: The conditions FIXEDOVERFLOW and SIZE cannot occur together, because the same hardware condition raises both of them.
2. Hardware exceptions in the order:
   data
   specification
   addressing
   protection

   Conditions and exceptions are raised in the above order until one of the following situations occurs:
   
   • A GO TO statement in an on-unit is executed. All other exceptions will then be lost.
   
   • The ERROR condition is raised. If the program is terminated as a result of this action (that is, system action causing the ERROR condition to be raised, followed by the FINISH condition), messages will be printed to indicate the nature of the unprocessed exceptions. The exceptions themselves will not be processed.

   When an interrupt results from multiple exceptions, only one of the PL/I conditions is raised for each type of exception that occurred.

   When a multiple-exception imprecise interrupt occurs, the ONCOUNT built-in function provides a binary integer count of the number of exception types (that have PL/I on-conditions associated with them) that remain to be processed. If the ONCOUNT built-in function is used when only a single exception has occurred, or if it is used outside an on-unit, a count value of zero is indicated.
Appendix E: Shared Library Cataloged Procedures

The shared library is a PL/I facility that allows an installation to load PL/I resident library modules into the link pack area (LPA) so that they are available to all PL/I programs. This reduces space overheads.

The resident library subroutines to be included in the shared library can be chosen by the installation; they must include the initialization routine, the error-handling routine, the open file routine, and all modules addressed from the TCA that are not identical for multitasking and non-multitasking programs. Further details of the shared library are given in the publications PL/I Library Cataloged Procedures, PL/I Optimizing Compiler: Execution Logic and OS PL/I Optimizing Compiler: System Information.

The routines in the shared library are held in two of three link-pack-area modules: IBMBPSM, and either IBMBPSL or its multitasking equivalent IBMTPSL. Each of the link-pack modules contains a number of library routines, and is headed by an addressing control block known as a transfer vector. IBMBPSM contains those modules in the shared library that are common to both multitasking and non-multitasking PL/I environments. IBMTPSL contains the non-multitasking versions of those modules that are not identical in multitasking and non-multitasking PL/I environments. This module has a multitasking counterpart, IBMTPSL, which holds the multitasking versions of such modules.

Two further modules are also involved in handling the shared library. These are the shared library addressing modules IBMBPSR and its multitasking counterpart IBMTPSR. One or other of these modules, each of which has the alias PLISHRE, is link-edited with compiled code and held in the program region: IBMBPSR for non-multitasking programs, or IBMTPSR for multitasking programs. IBMTPSR and its multitasking counterpart hold dummy entry points which duplicate the names of all entry points of modules within the shared library. References to such entry points in compiled code are resolved to the dummy entry points in IBMBPSR or IBMTPSR.

You can use the shared library by using standard IBM-supplied cataloged procedures and overriding the link-edit and loader procedure steps.

Execution when Using the Shared Library

Use of the shared library is specified by the linkage editor statement INCLUDE PLISHRE. PLISHRE is an alias for the program region modules IBMBPSR and IBMTPSR. The appropriate module will therefore be loaded by the linkage editor (IBMBPSR for non-multitasking programs; IBMTPSR for multitasking programs). All compiled code external references to shared library module entry points are then resolved to the dummy entry points in IBMBPSR (or IBMTPSR). Similarly WXTRNs in the program region module are resolved if compiled code issues an EXTRN for the entry point.

A load module created for use with one shared library will not execute with a different shared library. You will have to link-edit the object module again, including the dummy transfer vector module for the different shared library.

You must remember that the linkage editor or loader require a large amount of main storage for external symbol dictionary tables while processing the dummy transfer vector module. If you specify SIZE=200K in the PARM field of your EXEC statement for the linkage editor or loader (and use a region or partition of equivalent size), you will get sufficient main storage for processing with the largest possible shared library.

Your PL/I program may take slightly longer to execute when using a shared library, because all library calls have to pass through the transfer vectors. However, your main storage requirements for a region will be greatly reduced if you have carefully selected your shared library modules to suit the operating environment.

Multitasking Considerations

The shared library has been designed so that multitasking does not affect it. If PLI.TASK is specified before PLI.BASE, the linkage editor statement INCLUDE PLISHRE will result in the module IBMBPSR being loaded and linked in the program region. When control passes to the code following the IBMBPSR entry point in IBMTPSR, a request is made to the system to load the multitasking shared library module IBMTPSR. The program then runs in the usual manner.
with the multitasking modules.

An installation may specify that it does not require either the multitasking or the non-multitasking modules in the shared library. However both multitasking and non-multitasking versions of the program region module will still be created. The module for the unwanted environment will be a dummy. This prevents problems should an INCLUDE PLISHRE statement be included in a program that is intended to run in the environment with no shared library. If this process was not carried out, such a statement could result in the incorrect environment being initialized.

USING STANDARD IBM CATALOGED PROCEDURES

Standard IBM-supplied cataloged procedures that use the linkage editor or loader (see Chapter 11) can be used to specify the shared library. This is done by overriding the SYSLIN DD statement in the link-edit or load-and-go procedure steps to ensure that the shared library addressing module IBMBPSR is the first module to be included by the linkage editor or loader and that its entry point in the resulting load module has the name PLISHRE. For example, the cataloged procedure PLIXCL requires the following statements to make use of the shared library.

```
//STEP1 EXEC PLIXCL
//LINED.SYSIN DD *
   INCLUDE SYSLIB (PLISHRE)

   (add further input here)
   /*
```

You can add other linkage-editor control statements by placing them as indicated. For example, to give the resulting load module the name MINE, add the statement:

```
NAME MINE (R)
```

between the ENTRY and /* statements.
Appendix F: Programming Example

This appendix, consisting of a PL/I sample program, illustrates all the components of the listings produced by the compiler and the linkage editor. The listings themselves are described in Chapters 4 and 5.

The function of the program is fully documented in both the preprocessor input and the source listing by means of PL/I comments. These comments consist of lines of text each preceded by /* and followed by */. Note that the /* must not appear in columns 1 and 2 of the input record because it will be taken as a job control delimiter statement.

Most pages of the listings contain brief notes explaining the contents of the pages.

TRANSIENT LIBRARY MODULES IN THE LINK PACK AREA

Any module in the PL/I Transient Library may be placed in the link pack area (LPA) without any change in procedure.
OPTIONS SPECIFIED

AG,A,C,ESD,GS,LST,M,MAP,OP,IS,STG,SYN,X,MAP(2,72,1),LC(55)

OPTIONS USED

AGGREGATE   NOCOUNT   CHARSET(60, EBCDIC)
ATTRIBUTES   NODECK    FLAG(1)
COMPILE      NOFLOW    LINECOUNT(55)
ESD          NONGROUP   MARGINS(2, 72, 1)
GOSTMT       NOTIMPRECISE OPTIMIZE(TIME)
INSOURCE     NOINCLUDE  SEQUENCE(73, 80)
LIST         NOMARGINI  SIZE(101584)
MESSAGE      NOMDECK    
MACRO        NONUMBER   
MAP          NOTERMINAL 
NEST         
OBJECT       
OFFSET       
OPTIONS      
SOURCE       
STMT         
STORAGE      
SYNTAX       
XREF         

Start of the compiler listing.

List of options specified in the PARM parameter of the EXEC statement.

List of options used, whether obtained by default, or by being specified explicitly.
PREPROCESSOR INPUT

LINE 00269040
1 /***** PL/I SAMPLE PROGRAM. *****/ 00269050
2 */%*************************************************************************/ 00269060
3 /* USES COMPILE-TIME PREPROCESSOR TO MODIFY PL/I (F) SOURCE FOR */ 00269070
4 /* USE WITH THIS COMPILER. THE PREPROCESSOR STATEMENTS FOLLOWING */ 00269080
5 /* COULD BE PLACED ON A LIBRARY AND USED TO MODIFY SEVERAL SOURCE */ 00269090
6 /* PROGRAMS BY MEANS OF THE PREPROCESSOR INCLUDE STATEMENT. THEY */ 00269100
7 /* PERFORM THE FOLLOWING FUNCTIONS: */ 00269110
8 /* */ 00269120
9 /* 1. CONVERT CALLS TO FOLLOWING PL/I (F) IHE... ROUTINES TO THE */ 00269130
10 /* EQUIVALENT NEW PLI... ROUTINES- */ 00269140
11 /* IHEDUMP/J/C/T TO PLIDUMP, */ 00269150
12 /* IHESRAT/B/C/D TO PLISRAT/B/C/D, */ 00269160
13 /* IHECKP/S/T TO PLICKPT, */ 00269170
14 /* IHEREXT/T TO PLIREXT/PLIRENC, */ 00269180
15 /* IHERESRC/IHERET/A TO PLIRET2. */ 00269190
16 /* */ 00269200
17 /* 2. CHANGE FIRST DECLARE/DCL STATEMENT FOUND TO INCLUDE */ 00269210
18 /* BUILTIN ATTRIBUTE FOR FOLLOWING BUILT-IN FUNCTIONS:(WHICH */ 00269220
19 /* DO NOT TAKE ARGUMENTS, AND SO ARE NOT IMPLICITLY DECLARED */ 00269230
20 /* BUILTIN FOR THIS COMPILER - AS THEY WOULD BE FOR PL/I (F))- */ 00269240
21 /* DATE, TIME, ONCODE, ONCHRN, ONSOURCE, ONLOC, */ 00269250
22 /* ONFILE, ONKEY, EMPTY, NULL. */ 00269260
23 /* NOTE: THE ONCOUNT IF IS OMITTED FROM THIS LIST, & IS USED */ 00269270
24 /* LATER TO SHOW THE EFFECT OF NOT DECLARING IT BUILTIN. */ 00269280
25 /* ANY REFERENCES TO IHE--- ROUTINES MUST BE REMOVED */ 00269290
26 /* FROM DECLARE STATEMENTS BEFORE THE SOURCE PROGRAM IS */ 00269300
27 /* PREPROCESSED. OTHERWISE FAILURES MAY OCCUR WHEN THE */ 00269310
28 /* CONVERTED PROGRAM IS LINK-EDITED. */ 00269320
29 /* */ 00269330
30 /* 3. CHANGE 'NULL0' TO 'NULL' - THERE IS NO NULL0 BUILTIN */ 00269340
31 /* FUNCTION FOR THIS COMPILER; NULL MUST BE USED BOTH WITH */ 00269350
32 /* POINTER AND OFFSET VARIABLES. */ 00269360
33 /* */ 00269370
34 /* */ 00269380
35 /**************************************************************************/
<table>
<thead>
<tr>
<th>LINE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>36</td>
<td>% DCL (IHEDUMP, IHEDUMJ, IHEDUMC, IHEDUMT, DECLARE, DCL, IHECKPT, IHECKPS) ENTRY;</td>
</tr>
<tr>
<td>37</td>
<td>% 00269390</td>
</tr>
<tr>
<td>38</td>
<td>% DCL (IHESRTA, IHESRTB, IHESRTC, IHESRTD, IHEREST, IHERESN, IHESA.RC, IHETSAC, NULLO) CHAR;</td>
</tr>
<tr>
<td>39</td>
<td>% 00269410</td>
</tr>
<tr>
<td>40</td>
<td>% DCL COUNT FIXED;</td>
</tr>
<tr>
<td>41</td>
<td>% 00269420</td>
</tr>
<tr>
<td>41</td>
<td>% COUNT = 0 /* FIRST-TIME-IN SWITCH. */ 00269440</td>
</tr>
<tr>
<td>42</td>
<td>% DEACTIVATE DECLARE, DCL /* ENSURE MODIFIED STATEMENTS */ 00269450</td>
</tr>
<tr>
<td>43</td>
<td>% ACTIVATE DECLARE, /* ARE NOT RESCA.NNED DURIN3 */ 00269460</td>
</tr>
<tr>
<td>44</td>
<td>% DCL NORESCAN /* PREPROCESSOR REPLACEMENT. */ 00269470</td>
</tr>
</tbody>
</table>
DECLARE: DCL: /* GENERATE BUILTIN DECLARES. */ 00269480
PROC RETURNS(CHAR); 00269490
COUNT = COUNT + 1 /* COUNT = 1 IF 1ST TIME IN. */ 00269500
IF COUNT = 1 00269510
THEN RETURN('DCL (DATE,TIME,ONCHAR,ONSENSE,ONCODE, ' ||
'CKPT_RETC FIXED BIN(31),''); 00269520
ELSE RETURN('DCL'); 00269530
END; 00269540

DECLARE: OCL: /* GENERATE BUILTIN DECLARES. */ 00269550
PROC RETURNS(CHAR): COUNT = COUNT + 1 /* COUNT = 1 IF 1ST TIME IN. */ 00269560
IF COUNT = 1 THEN RETURN('OCL (DATE,TIME,ONCHAR,ONSENSE,ONCODE, '
'ONLOC,ONFILE,ONKEY,EMPTY,NULL) BUILTIN, ,
'CKPT RETC FIXED BIN(31),''); 00269570
ELSE RETURN('OCL'); 00269580
END; 00269590

DECLARE: IDHE: /* REPLACED BY CALL TO */ 00269600
PROC(ID#) RETURNS(CHAR): /* PLIDUMP ROUTINE, INCLUDING */ 00269610
DCL ID# CHAR /* ORIGINAL ID(IF PRESENT). */ 00269620
IF ID# = '' THEN RETURN('PLIDUMP'); 00269630
ELSE RETURN('PLIDUMP("TFCA",'''''' ID# '''''' ' '); 00269640
END; 00269650

DECLARE: ARGS: /* CHANGE TO */ 00269660
PROC(ARG1, ARG2, ARG3, ARG4): /* DEFAULTS GENERATED WHERE */ 00269670
RETURNS(CHAR): /* NO ARGUMENTS ORIGINALLY. */ 00269680
DCL (ARG1, ARG2, ARG3, ARG4) CHAR; 00269690
IF ARG1 = '' THEN ARG1 = ''SYSCHK''; 00269700
IF ARG2 = '' THEN ARG2 = ''; 00269710
IF ARG3 = '' THEN ARG3 = ''PS''; 00269720
IF ARG4 = '1 THEN ARG4 = ' CKPT RETC'; 00269730
RETURN('PLICKPT(' ARG1 || ', ' ARG2 || ', ' ARG3 || ', ' ARG4 || ')'); 00269740
END; 00269750

DECLARE: ARGS: /* REPLACE */ 00269760
IHESRTA = 'PLISRTA' /* CALLS TO */ 00269770
IHESRTB = 'PLISRTB' /* IHE--- */ 00269780
IHESRTC = 'PLISRTC' /* ROUTINES */ 00269790
IHESRTD = 'PLISRTD' /* BY */ 00269800
IHESRN = 'PLICANC' /* CALLS TO */ 00269810
IHESRC = 'PLIREC' /* PLI--- */ 00269820
IHETSAC = 'PLIRETC' /* ROUTINES. */ 00269830

DECLARE: IDNULL: /* THERE IS NO NULLO BUILTIN FUNCTION FOR THIS */ 00269840
PROC: /* NULL MUST BE USED INSTEAD. */ 00269850
NULO = 'NULL'; 00269860
/* END OF PREPROCESSOR STATEMENTS; SOURCE STATEMENTS FOLLOW HERE: */ 00269850

SAMPLE:
PROC OPTIONS(MAIN);
DECLARE (PDATE, PTIME) CHAR(6);
DECLARE CVAR CHAR(255) VAR;
DCL 1 BINVAR,
  2 RETCODE FIXED BIN(31,0),
  2 FBVAR FIXED BIN;
PDATE = DATE;
PTIME = TIME;
PUT SKIP EDIT('SAMPLE PROGRAM: DATE = ', PDATE, ' TIME = ',
RETCODE = 0101;
ON ERROR BEGIN:
CALL IHEDUMP;
CALL IHEDUMC(RETCODE);
CALL IHEDUMT;
FBVAR = ONCODE;
CVAR = ONCOL;
CVAR = ONLOC;
CVAR = ONFILE;
CVAR = ONKEY;
*/ 00269970
PL/I OPTIMIZING COMPILER

****** PL/I SAMPLE PROGRAM. *****

LINE
109  /* THIS STATEMENT, WHICH WILL NEVER BE EXECUTED, USES 'ONCOUNT' WHICH IS NEITHER EXPLICITLY NOR IMPLICITLY DECLARED BUILTIN. THE EFFECT IS SHOWN IN THE ATTRIBUTE LISTING AND DIAGNOSTIC MESSAGES. */
110     FBVAR = ONCOUNT;
111     END;

112  /* THIS IS A DUMMY PROCEDURE TO ILLUSTRATE OTHER PREPROCESSOR REPLACEMENTS/NON-IMPLICITLY DECLARED BUILTIN FUNCTIONS. IT WILL NEVER BE EXECUTED. */
113     DUMMY:
114       PROC;
115       DCL AVAR AREA BASED(PVAR),
116       OVAR OFFSET(AVAR),
117       A ENTRY RETURNS(CHAR(80)),
118       SIZE FIXED BIN(31,0);
119       AVAR = EMPTY;
120       PVAR = NULL;
121       OVAR = NULL;
122       CALL IHESRTA('ARG1', 'ARG2', SIZE, RETCODE); /* S */
123       CALL IHESRTB('ARG1', 'ARG2', SIZE, RETCODE, A); /* O */
124       CALL IHESRTC('ARG1', 'ARG2', SIZE, RETCODE, B); /* R */
125       CALL IHESRTD('ARG1', 'ARG2', SIZE, RETCODE, A, B); /* T */
126       CALL IHESRTF('ARG1', 'ARG2', 'PS', RETCODE); /* CHECKPOINT */
127       CALL IHESRTG; /* CHECKPOINT */
128       CALL IHESRT;
129       CALL IHESRT;
130       CALL IHESRT;
131       CALL IHESRT;
132       CALL IHESRT;
133       CALL IHESRT;
134       CALL IHESRT(RETCODE); /* SET RETURN CODE(TASKING) */
135       A:    PROC RETURNS(CHAR(80)); END; /* DUMMY EXIT */
136       /* PROCEDURES */
137       B:    PROC(RECORD); DCL RECORD CHAR(80); END; /* FOR SORT. */
138       END DUMMY;
139       CALL IHESARC(RETCODE); /* SET RETURN CODE(NONTASKING) */
140       PUT SKIP LIST('END SAMPLE PROGRAM');
141       END SAMPLE;
## Preprocessor Diagnostic Messages

<table>
<thead>
<tr>
<th>ERROR ID</th>
<th>LINE</th>
<th>Message Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IEL0217I E 97</td>
<td></td>
<td>Missing left parenthesis from argument list for procedure 'IHEDUMP'. Procedure invoked without arguments.</td>
</tr>
<tr>
<td>IEL0217I E 102</td>
<td></td>
<td>Missing left parenthesis from argument list for procedure 'IHEDUMT'. Procedure invoked without arguments.</td>
</tr>
<tr>
<td>IEL0217I E 131</td>
<td></td>
<td>Missing left parenthesis from argument list for procedure 'IHECKPT'. Procedure invoked without arguments.</td>
</tr>
</tbody>
</table>

## Warning Diagnostic Messages

<table>
<thead>
<tr>
<th>ERROR ID</th>
<th>LINE</th>
<th>Message Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IEL01841 W 97</td>
<td></td>
<td>Too few arguments to function 'IHEDUMP'. NULL strings passed as missing arguments.</td>
</tr>
<tr>
<td>IEL01841 W 102</td>
<td></td>
<td>Too few arguments to function 'IHEDUMT'. NULL strings passed as missing arguments.</td>
</tr>
<tr>
<td>IEL01841 W 131</td>
<td></td>
<td>Too few arguments to function 'IHECKPT'. NULL strings passed as missing arguments.</td>
</tr>
</tbody>
</table>

---

Diagnostic messages generated by the preprocessor. All messages generated by the optimizing compiler (including the preprocessor) are documented in the publication OS Optimizing Compiler: Messages.

1. "ERROR ID" This identifies the message as originating from the optimizing compiler (IEL), and gives the message number.
2. "L" This is the severity level of the message.
3. "LINE" This gives the number of the line in which the error occurred.
/* THIS SOURCE LISTING IS THE OUTPUT FROM THE PL/I PREPROCESSOR AND THE INPUT TO THE COMPILER. ALL THE PREPROCESSOR STATEMENTS HAVE BEEN EXECUTED AND ALL PREPROCESSOR COMMENTS HAVE BEEN DELETED */

STATEMENT NESTING LEVELS:
1

LINE NUMBERS Brought forward from the preprocessor input:
2

MAXIMUM DEPTH OF REPLACEMENT:
3

"E" in this column indicates that an error has occurred during a replacement attempt:
4

Source listing. This is the output from the preprocessor and the input to the compiler. All the preprocessor statements have been executed and all preprocessor comments have been deleted.

<table>
<thead>
<tr>
<th>Statement Number</th>
<th>Source Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PROC OPTIONS(MAIN);</td>
</tr>
<tr>
<td>2</td>
<td>DCL (DATE, TIME, ONCHAR, ONSOURCE, ONCODE, ONLOC, ONFILE, ONKEY, EMPTY, NULL) BUILTIN, CRP_RET CHAR(6);</td>
</tr>
<tr>
<td>3</td>
<td>DCL CVAR CHAR(255) VAR;</td>
</tr>
<tr>
<td>4</td>
<td>DCL 1 BINVAR,</td>
</tr>
<tr>
<td></td>
<td>2 RETCODE FIXED BIN(31,0),</td>
</tr>
<tr>
<td></td>
<td>2 FBVAR FIXED BIN;</td>
</tr>
<tr>
<td>5</td>
<td>PDATE = DATE;</td>
</tr>
<tr>
<td>6</td>
<td>PTIME = TIME;</td>
</tr>
<tr>
<td>8</td>
<td>RETCODE = 0101;</td>
</tr>
<tr>
<td>9</td>
<td>ON ERROR BEGIN;</td>
</tr>
<tr>
<td>10</td>
<td>CALL PLIDUMP;</td>
</tr>
<tr>
<td>11</td>
<td>CALL PLIDUMP('TPCA','127');</td>
</tr>
<tr>
<td>12</td>
<td>CALL PLIDUMP('TPCA','RETCODE');</td>
</tr>
<tr>
<td>13</td>
<td>CALL PLIDUMP;</td>
</tr>
<tr>
<td>14</td>
<td>FBVAR = ONCODE;</td>
</tr>
<tr>
<td>15</td>
<td>CVAR = ONCHAR;</td>
</tr>
<tr>
<td>16</td>
<td>CVAR = ONSOURCE;</td>
</tr>
<tr>
<td>17</td>
<td>CVAR = ONLOC;</td>
</tr>
<tr>
<td>18</td>
<td>CVAR = ONFILE;</td>
</tr>
<tr>
<td>19</td>
<td>CVAR = ONKEY;</td>
</tr>
</tbody>
</table>
THIS STATEMENT, WHICH WILL NEVER BE EXECUTED, USES 'ONCOUNT' WHICH IS NEITHER EXPLICITLY NOR IMPLICITLY DECLARED BUILTIN. THE EFFECT IS SHOWN IN THE ATTRIBUTE LISTING AND DIAGNOSTIC MESSAGES.

```
20 2 0  FBVAR = ONCOUNT;
21 2 0  END;
```

THIS IS A DUMMY PROCEDURE TO ILLUSTRATE OTHER PREPROCESSOR REPLACEMENTS/ NON-IMPLICITLY DECLARED BUILTIN FUNCTIONS. IT WILL NEVER BE EXECUTED.

```
22 1 0  DUMMY:
      PROC;
23 2 0  DCL AVAR AREA BASED(PVAR),
       OVAR OFFSET(AVAR),
       A ENTRY RETURNS(CHAR(80)),
       SIZE FIXED BIN(31,0);
24 2 0  AVAR = EMPTY;
25 2 0  PVAR = NULL;
26 2 0  OVAR = NULL;
27 2 0  CALL PLISRTA('ARG1', 'ARG2', SIZE, RETCODE);
28 2 0  CALL PLISRTB('ARG1', 'ARG2', SIZE, RETCODE, A);
29 2 0  CALL PLISRTC('ARG1', 'ARG2', SIZE, RETCODE, B);
30 2 0  CALL PLISRTD('ARG1', 'ARG2', SIZE, RETCODE, A, B);
31 2 0  CALL PLICKPT('ARG1', 'ARG2', 'PS', RETCODE);
32 2 0  CALL PLICKPT('SYSCHK', '', 'PS', CKPT_RETC);
33 2 0  CALL PLIREST;
34 2 0  CALL PLICANC;
35 2 0  CALL PLIRET(RETCODE);
36 2 0  A:  PROC RETURNS(CHAR(80));  END;
38 2 0  B:  PROC(RECORD);  DCL RECORD CHAR(80);  END;  /* FOR SORT. */
41 2 0  END DUMMY;
42 1 0  CALL PLIRETC(RETCODE);
43 1 0  PUT SKIP LIST('END SAMPLE PROGRAM');
44 1 0  END SAMPLE;
```
### Attribute and Cross-reference Table

<table>
<thead>
<tr>
<th>DCL No.</th>
<th>Identifier</th>
<th>Attributes and References</th>
</tr>
</thead>
<tbody>
<tr>
<td>36</td>
<td>A</td>
<td>ENTRY RETURNS(CHARACTER (80)) 28, 30</td>
</tr>
<tr>
<td>23</td>
<td>AVAR</td>
<td>BASED (PVAR) ALIGNED AREA (1000) 24</td>
</tr>
<tr>
<td>38</td>
<td>B</td>
<td>ENTRY RETURNS(DECIMAL /* SINGLE */ FLOAT (6)) 29, 30</td>
</tr>
<tr>
<td>4</td>
<td>BINVAR</td>
<td>AUTOMATIC /* STRUCTURE */</td>
</tr>
<tr>
<td>2</td>
<td>CKPT_RETC</td>
<td>AUTOMATIC UNALIGNED BINARY FIXED (31,0) 32</td>
</tr>
<tr>
<td>3</td>
<td>CVAR</td>
<td>AUTOMATIC UNALIGNED CHARACTER (255) VARYING 15, 16, 17, 18, 19</td>
</tr>
<tr>
<td>2</td>
<td>DATE</td>
<td>BUILTIN 5</td>
</tr>
<tr>
<td>22</td>
<td>DUMMY</td>
<td>ENTRY RETURNS(DECIMAL /* SINGLE */ FLOAT (6))</td>
</tr>
<tr>
<td>2</td>
<td>EMPTY</td>
<td>BUILTIN 24</td>
</tr>
<tr>
<td>2</td>
<td>FVAR</td>
<td>/* IN BINVAR */ AUTOMATIC ALIGNED BINARY FIXED (15,0) 14, 20</td>
</tr>
<tr>
<td>2</td>
<td>NULL</td>
<td>BUILTIN 25, 26</td>
</tr>
<tr>
<td>2</td>
<td>ONCHAR</td>
<td>BUILTIN 15</td>
</tr>
<tr>
<td>2</td>
<td>ONCODE</td>
<td>BUILTIN 14</td>
</tr>
<tr>
<td>2</td>
<td>ONCOUNT</td>
<td>AUTOMATIC ALIGNED DECIMAL /* SINGLE */ FLOAT (6) 20</td>
</tr>
<tr>
<td>2</td>
<td>ONFILE</td>
<td>BUILTIN 18</td>
</tr>
</tbody>
</table>

**Attributes and Cross-reference Table.**

1. Number of the statement in the source listing in which the identifier is explicitly declared.
2. Asterisks indicate an undeclared identifier; all of its attributes are implied or supplied by default.
3. All identifiers used in the program listed in alphabetic order.
4. Declared and default attributes are listed. This list also includes descriptive comments.
5. Cross references. These are the numbers of all other statements in which the identifier appears.
<table>
<thead>
<tr>
<th>DCL NO.</th>
<th>IDENTIFIER</th>
<th>ATTRIBUTES AND REFERENCES</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>ONKEY</td>
<td>BUILTIN 19</td>
</tr>
<tr>
<td>2</td>
<td>ONLOC</td>
<td>BUILTIN 17</td>
</tr>
<tr>
<td>2</td>
<td>ONSOURCE</td>
<td>BUILTIN 16</td>
</tr>
<tr>
<td>23</td>
<td>OVAR</td>
<td>AUTOMATIC ALIGNED OFFSET (AVAR) 26</td>
</tr>
<tr>
<td>2</td>
<td>PDATE</td>
<td>AUTOMATIC UNALIGNED CHARACTER (6) 5,7</td>
</tr>
<tr>
<td>*********</td>
<td>PLICANC</td>
<td>BUILTIN 34</td>
</tr>
<tr>
<td>*********</td>
<td>PLICKPT</td>
<td>BUILTIN 31,32</td>
</tr>
<tr>
<td>*********</td>
<td>PLIDUMP</td>
<td>BUILTIN 10,11,12,13</td>
</tr>
<tr>
<td>*********</td>
<td>PLIREST</td>
<td>BUILTIN 33</td>
</tr>
<tr>
<td>*********</td>
<td>PLIRETC</td>
<td>BUILTIN 42,35</td>
</tr>
<tr>
<td>*********</td>
<td>PLISRTA</td>
<td>BUILTIN 27</td>
</tr>
<tr>
<td>*********</td>
<td>PLISRTB</td>
<td>BUILTIN 28</td>
</tr>
<tr>
<td>*********</td>
<td>PLISRTC</td>
<td>BUILTIN 29</td>
</tr>
<tr>
<td>*********</td>
<td>PLISRTD</td>
<td>BUILTIN 30</td>
</tr>
<tr>
<td>2</td>
<td>PTIME</td>
<td>AUTOMATIC UNALIGNED CHARACTER (6) 6,7</td>
</tr>
<tr>
<td>*********</td>
<td>PVAR</td>
<td>AUTOMATIC ALIGNED POINTER 24,25</td>
</tr>
<tr>
<td>DCL NO.</td>
<td>IDENTIFIER</td>
<td>ATTRIBUTES AND REFERENCES</td>
</tr>
<tr>
<td>--------</td>
<td>------------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>39</td>
<td>RECORD</td>
<td>/* PARAMETER */ UNALIGNED CHARACTER (80)</td>
</tr>
<tr>
<td>4</td>
<td>RETCODE</td>
<td>/* IN BINVAR */ AUTOMATIC ALIGNED BINARY FIXED (31,0) 8,42 27,28,29,30,31,35</td>
</tr>
<tr>
<td>1</td>
<td>SAMPLE</td>
<td>EXTERNAL ENTRY RETURNS(DECIMAL /* SINGLE */ FLOAT (6))</td>
</tr>
<tr>
<td>23</td>
<td>SIZE</td>
<td>AUTOMATIC ALIGNED BINARY FIXED (31,0) 27,28,29,30</td>
</tr>
<tr>
<td></td>
<td>SYSPRINT</td>
<td>EXTERNAL FILE PRINT 7,43</td>
</tr>
<tr>
<td>2</td>
<td>TIME</td>
<td>BUILTIN 6</td>
</tr>
</tbody>
</table>
### Aggregate Length Table

<table>
<thead>
<tr>
<th>DCL NO.</th>
<th>IDENTIFIER</th>
<th>LWL</th>
<th>DIMS</th>
<th>OFFSET</th>
<th>ELEMENT LENGTH</th>
<th>TOTAL LENGTH</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>BINVAR</td>
<td>1</td>
<td>6</td>
<td>0</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>RETCODE</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>FBVAR</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

**SUM OF CONSTANT LENGTHS**: 6

---

Aggregate Length Table:

1. **Number of the statement in which the aggregate is declared, or, for a controlled aggregate, the number of the associated ALLOCATE statement.**

2. **The elements of the aggregate as declared.**

3. **Length of each element of the aggregate.**

4. **Sum of the lengths of aggregates whose lengths are constant.**
## Storage Requirements

<table>
<thead>
<tr>
<th>Block, Section or Statement</th>
<th>Type</th>
<th>Length (HEX)</th>
<th>DSA Size (HEX)</th>
</tr>
</thead>
<tbody>
<tr>
<td>*SAMPLE1</td>
<td>PROGRAM CSECT</td>
<td>2156</td>
<td>86C</td>
</tr>
<tr>
<td>*SAMPLE2</td>
<td>STATIC CSECT</td>
<td>788</td>
<td>314</td>
</tr>
<tr>
<td>SAMPLE</td>
<td>PROCEDURE BLOCK</td>
<td>454</td>
<td>1C6</td>
</tr>
<tr>
<td>9</td>
<td>ON UNIT</td>
<td>630</td>
<td>276</td>
</tr>
<tr>
<td>DUMMY</td>
<td>PROCEDURE BLOCK</td>
<td>834</td>
<td>342</td>
</tr>
<tr>
<td>A</td>
<td>PROCEDURE BLOCK</td>
<td>106</td>
<td>6A</td>
</tr>
<tr>
<td>B</td>
<td>PROCEDURE BLOCK</td>
<td>124</td>
<td>7C</td>
</tr>
</tbody>
</table>

### Description

Storage requirements. This table gives the main storage requirements for the program. These quantities do not include the main storage that will be required by the resident and transient library subroutines that will be included by the linkage editor or loaded dynamically during execution.

1. Name of the block, section, or number of the statement in the program.
2. Description of the block, section, or statement.
3. Length in bytes of the storage areas in both decimal and hexadecimal notation.
4. Length in bytes of the dynamic storage area (DSA) in both decimal and hexadecimal notation.
### External Symbol Dictionary

<table>
<thead>
<tr>
<th>SYMBOL</th>
<th>TYPE</th>
<th>ID</th>
<th>ADDR</th>
<th>LENGTH</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLISTART</td>
<td>SD</td>
<td>0001</td>
<td>000000</td>
<td>000044</td>
</tr>
<tr>
<td>*SAMPLE1</td>
<td>SD</td>
<td>0002</td>
<td>000000</td>
<td>000086C</td>
</tr>
<tr>
<td>*SAMPLE2</td>
<td>SD</td>
<td>0003</td>
<td>000000</td>
<td>0000314</td>
</tr>
<tr>
<td>PLITABS</td>
<td>WX</td>
<td>0004</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td>PLIXOPT</td>
<td>WX</td>
<td>0005</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td>PLIFLOW</td>
<td>WX</td>
<td>0006</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td>PLICOUNT</td>
<td>WX</td>
<td>0007</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td>IBMPIRA</td>
<td>ER</td>
<td>0008</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td>IBMPIRB</td>
<td>ER</td>
<td>0009</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td>IBMPIRC</td>
<td>ER</td>
<td>000A</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td>PLICALLA</td>
<td>LD</td>
<td>000006</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PLICALLB</td>
<td>LD</td>
<td>00000A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PLIMAIN</td>
<td>SD</td>
<td>000B</td>
<td>000000</td>
<td>000008</td>
</tr>
<tr>
<td>IBMBKCPA</td>
<td>ER</td>
<td>000C</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td>IBMBKCPA</td>
<td>ER</td>
<td>000D</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td>IBMBKCPB</td>
<td>ER</td>
<td>000E</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td>IBMBKCPA</td>
<td>ER</td>
<td>000F</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td>IBMKCPA</td>
<td>ER</td>
<td>0010</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td>IBMKSTD</td>
<td>ER</td>
<td>0011</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td>IBMKSTA</td>
<td>ER</td>
<td>0012</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td>IBMKSTC</td>
<td>ER</td>
<td>0013</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td>IBMKSTA</td>
<td>ER</td>
<td>0014</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td>IBMKSTA</td>
<td>ER</td>
<td>0015</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td>IBMKSTA</td>
<td>ER</td>
<td>0016</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td>IBMKSTA</td>
<td>ER</td>
<td>0017</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td>IBMKDMA</td>
<td>ER</td>
<td>0018</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td>IBMPRCA</td>
<td>ER</td>
<td>0019</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td>IELCNGA</td>
<td>SD</td>
<td>001A</td>
<td>000000</td>
<td>000072</td>
</tr>
<tr>
<td>IELCNGB</td>
<td>SD</td>
<td>001B</td>
<td>000000</td>
<td>000070</td>
</tr>
<tr>
<td>IBMSEDMA</td>
<td>ER</td>
<td>001C</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td>IBMSELMA</td>
<td>ER</td>
<td>001D</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td>IBMCCSA</td>
<td>ER</td>
<td>001E</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td>IBMCHFD</td>
<td>ER</td>
<td>001F</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td>IBMCDCE</td>
<td>ER</td>
<td>0020</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td>IBMCECID</td>
<td>ER</td>
<td>0021</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td>IBMSUID</td>
<td>ER</td>
<td>0022</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td>IBMBECOA</td>
<td>ER</td>
<td>0023</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td>IBMBEOLA</td>
<td>ER</td>
<td>0024</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td>IBMBJDTA</td>
<td>ER</td>
<td>0025</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td>IBMBJDTA</td>
<td>ER</td>
<td>0026</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td>IBMDOCLA</td>
<td>ER</td>
<td>0027</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td>IBMDOCLA</td>
<td>WX</td>
<td>0028</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td>IBMSEDB</td>
<td>WX</td>
<td>0029</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td>IBMSEBOA</td>
<td>ER</td>
<td>002A</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td>IBMBSSIOE</td>
<td>WX</td>
<td>002B</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td>IBMBSSIOE</td>
<td>WX</td>
<td>002C</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td>IBMBSSID</td>
<td>ER</td>
<td>002D</td>
<td>000000</td>
<td></td>
</tr>
</tbody>
</table>

**External symbol dictionary.**

1. "SYMBOL" A list of all the external symbols that make up the object module.
2. "TYPE" Type of external symbol as follows:
   - CM Common area.
   - ER External reference.
   - LD Label definition.
   - PR Pseudo-register.
   - SD Section definition.
   - WX Weak external reference.
3. "ID" All entries, except LD-type entries, are identified by a hexadecimal number.
4. "ADDR" Address (in hexadecimal) of LD-type entries only.
5. "LENGTH" Length in bytes (in hexadecimal) of LD, CM, and PR type entries only.
<table>
<thead>
<tr>
<th>IBMSPLA</th>
<th>ER</th>
<th>002E</th>
<th>000000</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBMSPDA</td>
<td>ER</td>
<td>002F</td>
<td>000000</td>
</tr>
<tr>
<td>IBMCKDD</td>
<td>ER</td>
<td>0030</td>
<td>000000</td>
</tr>
<tr>
<td>IBMSCCA</td>
<td>WX</td>
<td>0031</td>
<td>000000</td>
</tr>
<tr>
<td>IBMSCB</td>
<td>WX</td>
<td>0032</td>
<td>000000</td>
</tr>
<tr>
<td>IBMISTS</td>
<td>WX</td>
<td>0033</td>
<td>000000</td>
</tr>
<tr>
<td>SAMPLE</td>
<td>LD</td>
<td>0034</td>
<td>000000 000020</td>
</tr>
<tr>
<td>SYSPINT</td>
<td>SD</td>
<td>0034</td>
<td>000000 000020</td>
</tr>
</tbody>
</table>
PL/I OPTIMIZING COMPILER

STATIC INTERNAL STORAGE MAP

<table>
<thead>
<tr>
<th>Offset (hex)</th>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000000</td>
<td>LOCATOR</td>
<td>DCON</td>
<td>Section for the program. This control statement is the third standard entry in the external symbol dictionary.</td>
</tr>
<tr>
<td>0000000001</td>
<td>TEXT</td>
<td>DATE</td>
<td>Six-digit offset (in hexadecimal).</td>
</tr>
<tr>
<td>0000000002</td>
<td>COMMENT</td>
<td>LOCATOR</td>
<td>Text (in hexadecimal).</td>
</tr>
<tr>
<td>0000000003</td>
<td>CONSTANT</td>
<td>STORE</td>
<td>Comment indicating type of item to which the text refers. A comment text appears only against the first line of the text for an item.</td>
</tr>
</tbody>
</table>

(continued next page)
### PL/I Optimizing Compiler

```plaintext
/***** PL/I Sample Program. *****/

D9D6C7D9C1D47A40
CWC1E3C5407E40
0001D3
C840E3C9D4C5407E
40
0001DC
C5D5C440E2C1D4D7
D3C540D7D9D6C7D9
C1D4
0001EE
E3C6C3C1
0001F2
F1F2F7
0001F5
D9C5E3C3D6C4C5
0001FC
C1D9C7F1
000200
C1D9C7F2
000204
D7E2
000206
E2E8E2C3C8D2
00020C
000210
0C160000000001C8

#### Static External CSECTS

000000
FF00000C81201000
D1LCB
02D70F000000000
011300140008E2E8
E2D7D9C9D5E39040
```
**PL/I OPTIMIZING COMPILER**

/***** PL/I SAMPLE PROGRAM. *****/

<table>
<thead>
<tr>
<th>IDENTIFIER</th>
<th>LEVEL</th>
<th>OFFSET</th>
<th>(HEX)</th>
<th>CLASS</th>
<th>BLOCK</th>
</tr>
</thead>
<tbody>
<tr>
<td>BINVAR</td>
<td>192</td>
<td>C0</td>
<td>AUTO</td>
<td>SAMPLE</td>
<td></td>
</tr>
<tr>
<td>RETCODE</td>
<td>192</td>
<td>C0</td>
<td>AUTO</td>
<td>SAMPLE</td>
<td></td>
</tr>
<tr>
<td>FBVAR</td>
<td>196</td>
<td>C4</td>
<td>AUTO</td>
<td>SAMPLE</td>
<td></td>
</tr>
<tr>
<td>CVAR</td>
<td>224</td>
<td>E0</td>
<td>AUTO</td>
<td>SAMPLE</td>
<td></td>
</tr>
<tr>
<td>CKPT_RET</td>
<td>200</td>
<td>C8</td>
<td>AUTO</td>
<td>SAMPLE</td>
<td></td>
</tr>
<tr>
<td>PDATE</td>
<td>212</td>
<td>D4</td>
<td>AUTO</td>
<td>SAMPLE</td>
<td></td>
</tr>
<tr>
<td>PTIME</td>
<td>218</td>
<td>DA</td>
<td>AUTO</td>
<td>SAMPLE</td>
<td></td>
</tr>
<tr>
<td>GVAR</td>
<td>2</td>
<td>168</td>
<td>A8</td>
<td>AUTO</td>
<td>DUMMY</td>
</tr>
<tr>
<td>SIZE</td>
<td>2</td>
<td>172</td>
<td>AC</td>
<td>AUTO</td>
<td>DUMMY</td>
</tr>
<tr>
<td>PVAR</td>
<td>204</td>
<td>CC</td>
<td>AUTO</td>
<td>SAMPLE</td>
<td></td>
</tr>
<tr>
<td>ONCOUNT</td>
<td>208</td>
<td>D0</td>
<td>AUTO</td>
<td>SAMPLE</td>
<td></td>
</tr>
</tbody>
</table>
### TABLES OF OFFSETS AND STATEMENT NUMBERS

#### WITHIN PROCEDURE SAMPLE

<table>
<thead>
<tr>
<th>OFFSET (HEX)</th>
<th>STATEMENT NO.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>8E</td>
<td>5</td>
</tr>
<tr>
<td>A8</td>
<td>6</td>
</tr>
<tr>
<td>C0</td>
<td>7</td>
</tr>
<tr>
<td>150</td>
<td>8</td>
</tr>
<tr>
<td>158</td>
<td>9</td>
</tr>
<tr>
<td>15C</td>
<td>42</td>
</tr>
<tr>
<td>174</td>
<td>43</td>
</tr>
<tr>
<td>1AE</td>
<td>44</td>
</tr>
</tbody>
</table>

#### WITHIN ON UNIT

<table>
<thead>
<tr>
<th>OFFSET (HEX)</th>
<th>STATEMENT NO.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>56</td>
<td>10</td>
</tr>
<tr>
<td>60</td>
<td>11</td>
</tr>
<tr>
<td>A8</td>
<td>12</td>
</tr>
<tr>
<td>F0</td>
<td>13</td>
</tr>
<tr>
<td>FA</td>
<td>14</td>
</tr>
<tr>
<td>118</td>
<td>15</td>
</tr>
<tr>
<td>13E</td>
<td>16</td>
</tr>
<tr>
<td>180</td>
<td>17</td>
</tr>
<tr>
<td>1C4</td>
<td>18</td>
</tr>
<tr>
<td>206</td>
<td>19</td>
</tr>
<tr>
<td>248</td>
<td>20</td>
</tr>
<tr>
<td>266</td>
<td>21</td>
</tr>
</tbody>
</table>

#### WITHIN PROCEDURE DUMMY

<table>
<thead>
<tr>
<th>OFFSET (HEX)</th>
<th>STATEMENT NO.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>22</td>
</tr>
<tr>
<td>60</td>
<td>24</td>
</tr>
<tr>
<td>70</td>
<td>25</td>
</tr>
<tr>
<td>78</td>
<td>26</td>
</tr>
<tr>
<td>7C</td>
<td>27</td>
</tr>
<tr>
<td>D4</td>
<td>28</td>
</tr>
<tr>
<td>140</td>
<td>29</td>
</tr>
<tr>
<td>1AC</td>
<td>30</td>
</tr>
<tr>
<td>22C</td>
<td>31</td>
</tr>
<tr>
<td>298</td>
<td>32</td>
</tr>
<tr>
<td>2FE</td>
<td>33</td>
</tr>
<tr>
<td>308</td>
<td>34</td>
</tr>
<tr>
<td>312</td>
<td>35</td>
</tr>
<tr>
<td>32A</td>
<td>41</td>
</tr>
</tbody>
</table>

#### WITHIN PROCEDURE A

<table>
<thead>
<tr>
<th>OFFSET (HEX)</th>
<th>STATEMENT NO.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>36</td>
</tr>
<tr>
<td>56</td>
<td>37</td>
</tr>
</tbody>
</table>

#### WITHIN PROCEDURE B

<table>
<thead>
<tr>
<th>OFFSET (HEX)</th>
<th>STATEMENT NO.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>38</td>
</tr>
<tr>
<td>68</td>
<td>40</td>
</tr>
</tbody>
</table>
Object listing. This is a listing of the machine instructions generated by the optimizing compiler from the PL/I source program.

2. Assembler-language form of the machine instructions.
.....

PL/1 SAMPLE PROGRAM. • •••• /

MVI
MV1
MVI
LA
ST
MVC
BALR

169 (13) , X' 01'
0(13),X'CO'
1(13),X'24'
8,528(0,3)
8,92(0,13)
84(4,13),264(3)
2,0

• PROLOGUE BASE
000078 D2 07 D OBO 3 ODO

MVC

00007E
000082
000086

41 90 D OD4
50 90 D OBO
D2 07 D OB8 3 ODO

LA
ST
MVC

00008C
000090
000094

41 40 D ODA
50 40 D OB8
05 20

LA
ST
BALR

LOCATOR •• PDATE(8),
208 (3)
9,POATE
9,LOCATOR •• PDATE
LO~ATOR •• PTIME(8),
208 (3)
4,PTIME
4,LOCATOR •• PT1ME
2,0

PL/I OPTIMIZING COMPILER
00005C
000060
000064
000068
00006C
000070
000076

92
92
92
41
50
D2
05

01
CO
24
80
80
03
20

D
D
D
3
D
o

OA9
000
001
210
05C
054 3 108

/

• PROCEDURE BASE
• STATEMENT NUMBER 5
000096 41 40 D OD4
00009A 50 40 3 124
00009E 96 80 3 124
0000A2 41 10 3 124
0000A6 58 FO 3 068
OOOOAA 05 EF

LA
ST
01
LA
L
BALR

4,POATE
4,292(0,3)
292(3),X'80'
1,292(0,3)
15,A •• IBMBJDTA
14,15

• STATEMENT NUMBER 6
OOOOAe 41 40 D 200
OOOOBO 50 40 3 128
0000B4 96 80 3 128
0000B8 41 10 3 128
OOOOBC 58 FO 3 06e
OOOOco 05 EF
0000C2 D2 05 D ODA 0 200

LA
ST
01
LA
L
BALR
MVC

4,512(0,13}
4,296(0,3)
296 (3) ,X' 80'
1,296(0,3)
15,A •• IBMBJTTA
14,15
PT1ME(6),512(13)

OOOOEE
OOOOFO
0000F4
OOOOFA
OOOOFE
000100
000104
000108
00010C
000110
000114
000118
00011A
00011E
000122
000126
00012S
00012C
000132
000136
000138
00013e
000140
000144
000148
00014C
00014E
000152
000156

PAGE
05
58
D2
58
05
41
41
90
41
50
58
05
41
58
58
05
58
D2
58
05
41
50
41
50
58
05
58
58
05

67
40
16
70
67
EO
FO
EF
EO
EO
FO
EF
EO
10
70
67
60
08
70
67
EO
EO
EO
EO
FO
EF
10
FO
EF

BALR
L
MVC
L
BALR
LA
LA
STM
LA
ST
L
BALR
LA
L
L
BALR
L
MVC
L
BALR
LA
ST
LA
ST
L
BALR
L
L
BALR

6,7
4,520(0,13)
0(23,4),444(3)
7,A •• 1ELCGOB
6,7
14,176(0,13)
15, OED •• PDArE
14,15,0(1)
14,158 (0,3)
14,12(0,1)
15,A •• 1BMBSEDB
14,15
14,178(0,3)
1, 50 4 ( 0, 13)
7,A •• 1ELCGOA
6,7
6,520(0,13)
0(9,6),467(3}
7, A•• 1ELCG::>B
6,7
14,184 (0,13)
14,0(0,1}
14,182(0,3)
14,12(0,1)
15,A •• IBMBSEDB
14,15
1,504(0,13)
15,A •• 1BMBS1or
14,15

L

ST

15,272(0,3)
15,B1NVAR.RErCODE

• STATEMENr NUMBER 9
000160 92 oe D OA8

MV1

16S(13),X'OC'

• STATEMENT NUMBER 42
000164 41 40 0 oeo
000168 50 40 3 124
00016e 96 80 3 124
000170 IB 55
000172 41 10 3 124
000176 58 FO 3 138
00017A 05 EF

LA
ST
01
SR
LA
L
BALR

4,B1NVAR.RETCJDE
4,292(0,3)
292(3),X"80'
5,5
1,292(0,3)
15,312(0,3)
14,15

• STATEMENr NUMBER 43
00017C 41 1i0 D 200

LA

4,512(0,13)

D 208
4 000 3 lBC
3 048
0

3
1
3
1
3

OBO
09C
000
09E
OOC
078

3 OB2
D lFS
3 044
D 208
6 000 3 lD3
3 048
D
1
3
1
3

068
000
OB6
OOC
078

D 1F8
3 084

)II

Itj
Itj

CD

::s

g,.

....

w

..

t'IiI

I'd

• STATEMENr NUMBER 8
000158 58 FO 3 110
00015e 50 FO D oeo

ti

~

ti
I»

;:I

....=e
::s

\Q

~

w
I»
a

"C1

....
C\)

N
IN

....a

• STATEMENT NUMBER 7
0000C8 41 40 D 200
OOOOCC 50 40 3 130
OOOODO 92 20 D 211
0000D4 41 10 3 12C
000008 58 FO 3 080
OOOODC 05 EF
OOOODE 41 EO 3 098
0000E2 41 10 D 200
0000E6 50 10 D 1FS
OOOOEA 58 70 3 044

LA
ST
MV1
LA
L
BALR
LA
LA
ST
L

4,512(0,13)
4,304(0,3)
529(13),X'20'
1,300(0,3)
15,A •• 1BMBS1OE
14,15
14,152(0,3)
1,512(0,13)
1,504(0,13)
7, A•• 1ELCGOA

22


### PL/I OPTIMIZING COMPILER

#### PL/I SAMPLE PROGRAM.

**STATEMENT NUMBER 13**

<table>
<thead>
<tr>
<th>Statement</th>
<th>Line Numbers</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>0002B7</td>
<td>13</td>
<td>1,1</td>
</tr>
<tr>
<td>0002BA</td>
<td>13</td>
<td>5,5</td>
</tr>
<tr>
<td>0002BC</td>
<td>58 F0 3 13C</td>
<td>L 15,316(0,3)</td>
</tr>
<tr>
<td>0002C0</td>
<td>05 EF</td>
<td>BALR 14,15</td>
</tr>
</tbody>
</table>

**STATEMENT NUMBER 14**

<table>
<thead>
<tr>
<th>Statement</th>
<th>Line Numbers</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>0002C2</td>
<td>50 60 D 058</td>
<td>L 6,88(0,13)</td>
</tr>
<tr>
<td>0002C6</td>
<td>50 60 D 080</td>
<td>ST 6,176(0,13)</td>
</tr>
<tr>
<td>0002CA</td>
<td>41 40 6 0C4</td>
<td>LA 4,215(0,3)</td>
</tr>
<tr>
<td>0002CE</td>
<td>50 40 3 148</td>
<td>ST 6,328(0,3)</td>
</tr>
<tr>
<td>0002D7</td>
<td>96 80 3 148</td>
<td>DI 328(3),X<em>80</em></td>
</tr>
<tr>
<td>0002D6</td>
<td>41 10 3 148</td>
<td>LA 1,328(0,3)</td>
</tr>
<tr>
<td>0002DA</td>
<td>58 F0 3 060</td>
<td>L 15,A..IBMEOCA</td>
</tr>
<tr>
<td>0002DE</td>
<td>05 EF</td>
<td>BALR 14,15</td>
</tr>
</tbody>
</table>

**STATEMENT NUMBER 15**

<table>
<thead>
<tr>
<th>Statement</th>
<th>Line Numbers</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>0002E0</td>
<td>58 40 D 048</td>
<td>L 4,72(0,13)</td>
</tr>
<tr>
<td>0002E5</td>
<td>58 40 4 002</td>
<td>AH 4,2(0,4)</td>
</tr>
<tr>
<td>0002E6</td>
<td>91 40 4 006</td>
<td>TM 6(4),X<em>40</em></td>
</tr>
<tr>
<td>0002F0</td>
<td>47 80 2 0CA</td>
<td>BZ CL.16</td>
</tr>
<tr>
<td>0002F4</td>
<td>58 F0 4 010</td>
<td>L 15,16(0,4)</td>
</tr>
<tr>
<td>0002F8</td>
<td>48 70 3 0CA</td>
<td>LH 7,202(0,3)</td>
</tr>
<tr>
<td>0002FC</td>
<td>40 70 6 0EO</td>
<td>STH 7,CVAR</td>
</tr>
<tr>
<td>000300</td>
<td>D2 00 6 0E2 F 000</td>
<td>MVC CVAR+2(1),0(15)</td>
</tr>
</tbody>
</table>

**STATEMENT NUMBER 16**

<table>
<thead>
<tr>
<th>Statement</th>
<th>Line Numbers</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>000306</td>
<td>58 90 D 048</td>
<td>L 9,72(0,13)</td>
</tr>
<tr>
<td>000331</td>
<td>4A 90 9 002</td>
<td>AH 9,2(0,9)</td>
</tr>
<tr>
<td>00030E</td>
<td>58 90 9 000</td>
<td>CL.17 EQU *</td>
</tr>
<tr>
<td>000312</td>
<td>91 40 9 006</td>
<td>TM 6(9),X<em>40</em></td>
</tr>
<tr>
<td>000316</td>
<td>47 80 2 0F0</td>
<td>BZ CL.17</td>
</tr>
<tr>
<td>00031A</td>
<td>58 40 9 018</td>
<td>L 8,24(0,9)</td>
</tr>
<tr>
<td>00031E</td>
<td>48 80 9 01C</td>
<td>LH 8,28(0,9)</td>
</tr>
<tr>
<td>000322</td>
<td>41 E0 0 0FF</td>
<td>LA 14,255(0,0)</td>
</tr>
<tr>
<td>000326</td>
<td>19 E8</td>
<td>CR 14,8</td>
</tr>
<tr>
<td>000328</td>
<td>47 D0 2 110</td>
<td>BNH CL.18</td>
</tr>
<tr>
<td>000332</td>
<td>10 E8</td>
<td>LR 14,8</td>
</tr>
<tr>
<td>000332E</td>
<td>40 E0 6 0EO</td>
<td>STH 14,CVAR</td>
</tr>
<tr>
<td>000333</td>
<td>48 E0 3 0CA</td>
<td>SH 14,202(0,3)</td>
</tr>
<tr>
<td>000336</td>
<td>47 40 2 12A</td>
<td>BM CL.19</td>
</tr>
<tr>
<td>00033A</td>
<td>44 E0 2 124</td>
<td>EX 14,CL.20</td>
</tr>
<tr>
<td>00033E</td>
<td>47 F0 2 12A</td>
<td>B CL.21</td>
</tr>
</tbody>
</table>

**STATEMENT NUMBER 17**

<table>
<thead>
<tr>
<th>Statement</th>
<th>Line Numbers</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>000348</td>
<td>41 40 0 0CD</td>
<td>LA 4,192(0,13)</td>
</tr>
<tr>
<td>000350</td>
<td>96 80 3 14C</td>
<td>ST 4,332(0,3)</td>
</tr>
<tr>
<td>000354</td>
<td>41 10 3 14C</td>
<td>LA 1,332(0,3)</td>
</tr>
<tr>
<td>000355</td>
<td>58 40 0 0BC</td>
<td>L 4,192(0,13)</td>
</tr>
<tr>
<td>000362</td>
<td>48 80 0 0CA</td>
<td>LH 8,196(0,13)</td>
</tr>
<tr>
<td>000366</td>
<td>41 E0 0 0FF</td>
<td>LA 14,255(0,0)</td>
</tr>
<tr>
<td>00036A</td>
<td>19 E8</td>
<td>CR 14,8</td>
</tr>
<tr>
<td>000370</td>
<td>18 E8</td>
<td>RN 14,8</td>
</tr>
<tr>
<td>000372</td>
<td>40 E0 6 0EO</td>
<td>STH 14,CVAR</td>
</tr>
<tr>
<td>000376</td>
<td>4B E0 3 0CA</td>
<td>SH 14,202(0,3)</td>
</tr>
<tr>
<td>00037A</td>
<td>47 E0 2 16E</td>
<td>BM CL.23</td>
</tr>
<tr>
<td>00037E</td>
<td>44 E0 2 168</td>
<td>EX 14,CL.24</td>
</tr>
<tr>
<td>000382</td>
<td>47 E0 2 16E</td>
<td>CL.25 EQU *</td>
</tr>
<tr>
<td>000386</td>
<td>D2 00 6 0E2 4 000</td>
<td>MVC CVAR+2(1),0(4)</td>
</tr>
</tbody>
</table>

**STATEMENT NUMBER 18**

<table>
<thead>
<tr>
<th>Statement</th>
<th>Line Numbers</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>00038C</td>
<td>58 40 D 048</td>
<td>L 4,72(0,13)</td>
</tr>
<tr>
<td>000390</td>
<td>4A 40 4 002</td>
<td>AH 4,2(0,4)</td>
</tr>
<tr>
<td>000394</td>
<td>58 40 4 000</td>
<td>L 4,0(0,4)</td>
</tr>
<tr>
<td>000398</td>
<td>91 80 4 006</td>
<td>TM 6(4),X<em>80</em></td>
</tr>
<tr>
<td>00039C</td>
<td>47 80 2 176</td>
<td>BZ CL.26</td>
</tr>
<tr>
<td>0003A0</td>
<td>58 E0 4 008</td>
<td>L 14,8(0,4)</td>
</tr>
<tr>
<td>0003A4</td>
<td>48 90 4 00C</td>
<td>LH 9,120(0,4)</td>
</tr>
<tr>
<td>0003A8</td>
<td>41 40 0 0FF</td>
<td>LA 4,255(0,0)</td>
</tr>
<tr>
<td>0003AC</td>
<td>19 49</td>
<td>CR 4,9</td>
</tr>
<tr>
<td>0003AE</td>
<td>47 D0 2 196</td>
<td>BNH CL.27</td>
</tr>
<tr>
<td>0003B2</td>
<td>18 49</td>
<td>LR 4,9</td>
</tr>
<tr>
<td>0003B4</td>
<td>40 40 6 0EO</td>
<td>STH 4,CVAR</td>
</tr>
<tr>
<td>0003B8</td>
<td>48 40 3 0CA</td>
<td>SH 4,202(0,3)</td>
</tr>
<tr>
<td>0003BC</td>
<td>47 40 2 180</td>
<td>BM CL.28</td>
</tr>
<tr>
<td>0003C0</td>
<td>44 40 2 1AA</td>
<td>EX 4,CL.29</td>
</tr>
<tr>
<td>0003C4</td>
<td>47 F0 2 180</td>
<td>B CL.30</td>
</tr>
<tr>
<td>0003C8</td>
<td>D2 00 6 0E2 F 000</td>
<td>MVC CVAR+2(1),0(14)</td>
</tr>
</tbody>
</table>
PL/I OPTIMIZING COMPILER

***** PL/I SAMPLE PROGRAM. *****

0003CE
0003CE

* STATEMENT NUMBER 19

0003CE 58 40 D 048
0003D2 4A 40 4 002
0003D6 58 40 4 000
0003DA 91 10 4 006
0003E2 58 80 2 188
0003E6 48 90 4 024
0003EA 41 40 0 0FF
0003EE 19 49
0003FO 47 40

* STATEMENT NUMBER 20

000400 78 00 6 0D0
000419 7E 00 3 114
000419 70 00 D 0C0
00042B 4E 40 2 1EC
00042F 47 40 2 1IP
0004A0 44 00 4 024
0004A0 D2 00 6 02E E 000
0004A4 CL.33 EQU *
0004A4 CL.35 EQU *

* STATEMENT NUMBER 21

00042E 18 0D 0
000430 58 D0 D 004
000434 58 E0 D 00C
000438 98 2C D 01C
00043C 05 1E

* ON UNIT BLOCK END

00043E 07 07

* STATEMENT NUMBER 22

000440
000447

* PROCEDURE BASE

000448 90 EC D 00C
00044C 47 F0 F 014
000450 00000000
00045A 00000000
000458 00000000

* STATEMENT NUMBER 23

000468 00 00

* STATEMENT NUMBER 24

00046A 55 00 C 00C
00046E 47 D0 F 030
000472 58 FO C 074
000476 05 EF
000478 58 EO D 048
00047C 18 FO
00047E 90 EO 1 048
000482 50 D0 1 004
000486 41 D1 0 000
00048A 50 50 D 058
00048E 92 80 D 000
000492 92 24 D 001
000496 B2 03 D 054 3 108
00049C 05 20

* STATEMENT NUMBER 25

000488 58 70 6 0CC
00048A 92 00 7 000
00048C 50 60 7 004

* STATEMENT NUMBER 26

00048E 92 24 D 001
000496 B2 03 D 054 3 108
00049C 05 20

PAGE 25
PL/I OPTIMIZING COMPILER

***** PL/I SAMPLE PROGRAM. *****

0004C0 50 70 D 0A8
ST 7,0VAR
0005B2 58 F0 3 178
L 15,376 (0,3)
0005B6 05 EF
BALR 14,15

* STATEMENT NUMBER 27
0004C4 2D 03 D 0C0 3 1PC
MVC 192 (4,13), 508 (3)
0004CA 2D 07 D 0C4 3 0E0
MVC 196 (8,13), 224 (3)
0004D0 41 80 D 0C0
LA 8,192 (0,13)
0004D4 41 80 D 0C4
ST 8,196 (0,13)
0004D8 41 40 D 0C4
LA 4,196 (0,13)
0004DC 40 3 150
ST 4,336 (0,3)
0004E0 2D 03 D 0CC 3 200
MVC 204 (4,13), 512 (3)
0004E6 2D 07 D 0DO 3 0E0
MVC 208 (8,13), 224 (3)
0004EC 41 40 D 0CC
LA 4,204 (0,13)
0004F0 40 40 D 0DO
ST 4,208 (0,13)
0004F4 41 40 D 0DO
LA 4,208 (0,13)
0004F8 40 3 154
ST 4,308 (0,3)
0004FC 41 40 D 0AC
LA 4,308 (0,3)
000500 40 3 158
ST 4,308 (0,3)
000504 41 40 6 0C0
LA 4,316 (0,3)
000508 40 3 15C
ST 4,316 (0,3)
00050C 96 80 3 15C
OI 308 (3), X'80'
000510 18 55
SR 5,5
000512 41 10 3 150
LA 1,336 (0,3)
000516 58 F0 3 160
L 15,352 (0,3)
00051A 05 EF
BALR 14,15

* STATEMENT NUMBER 28
00051C 2D 03 D 0C0 3 1PC
MVC 192 (4,13), 508 (3)
000522 2D 07 D 0C4 3 0E0
MVC 196 (8,13), 224 (3)
000529 41 80 D 0C0
LA 8,192 (0,13)
00052C 50 80 D 0C4
ST 8,196 (0,13)
000530 41 40 D 0C4
LA 4,196 (0,13)
000534 50 40 3 164
ST 4,356 (0,3)
000538 2D 03 D 0CC 3 200
MVC 204 (4,13), 512 (3)
00053E 2D 07 D 0DO 3 0E0
MVC 208 (8,13), 224 (3)
000544 41 80 D 0CC
LA 4,204 (0,13)
000548 50 80 D 0DO
ST 4,208 (0,13)
00054C 41 40 D 0DO
LA 4,208 (0,13)
000550 40 3 168
ST 4,360 (0,3)
000554 41 40 D 0AC
LA 4,360 (0,3)
000558 50 40 3 16C
ST 4,360 (0,3)
00055C 41 40 6 0C0
LA 4,360 (0,3)
000560 40 3 170
ST 4,368 (0,3)
000564 50 80 D 0DC
ST 13,220 (0,13)
000568 58 F0 3 024
L 15,36 (0,3)
00056C 50 40 3 16B
ST 15,364 (0,3)
000570 41 80 D 0DB
ST 4,372 (0,3)
000574 50 40 3 174
ST 4,372 (0,3)
000578 96 80 3 174
OI 372 (3), X'80'
00057E 18 55
SR 5,5
000580 41 10 3 164
LA 1,356 (0,3)

* STATEMENT NUMBER 30
000584 2D 03 D 0C0 3 1PC
MVC 192 (4,13), 508 (3)
000588 2D 07 D 0C4 3 0E0
MVC 196 (8,13), 224 (3)
000594 41 80 D 0C0
LA 8,192 (0,13)
000598 41 40 D 0C4
ST 8,196 (0,13)
0005A0 50 40 3 164
ST 4,356 (0,3)
0005A4 2D 03 D 0CC 3 200
MVC 204 (4,13), 512 (3)
0005A8 2D 07 D 0DO 3 0E0
MVC 208 (8,13), 224 (3)
0005B0 50 80 D 0DC
ST 14,208 (0,13)
0005B4 50 40 3 174
ST 4,372 (0,3)
0005B8 50 80 D 0DD
ST 4,372 (0,3)
0005C0 50 3 178
ST 4,372 (0,3)
0005C4 50 80 D 0CC
ST 4,372 (0,3)
0005CA 50 40 3 16B
ST 4,372 (0,3)
0005D0 50 80 D 0DO
ST 15,360 (0,3)
0005DC 50 80 D 0DC
ST 15,364 (0,3)
0005E0 50 40 3 174
ST 4,372 (0,3)
0005E4 50 80 D 0DD
ST 4,372 (0,3)
0005EF 96 80 3 174
ST 4,372 (0,3)
0005F2 05 EF
BALR 14,15
0005F4 41 40 D 0AC
LA 4,372 (0,3)
0005F8 41 40 D 0AC
LA 4,372 (0,3)
0005FC 41 40 D 0AC
LA 4,372 (0,3)
000600 41 40 D 0C4
ST 4,372 (0,3)
000604 50 80 D 0CC
ST 4,372 (0,3)
000608 41 40 D 0DO
ST 4,372 (0,3)
00060C 50 80 D 0DC
ST 4,372 (0,3)
000610 41 40 D 0DB
ST 4,372 (0,3)
000614 50 80 D 0DO
ST 4,372 (0,3)
000618 50 80 D 0DC
ST 4,372 (0,3)
00061C 41 40 D 0CC
ST 4,372 (0,3)
000620 50 80 D 0DD
ST 4,372 (0,3)
000624 41 40 D 0DO
ST 4,372 (0,3)
000628 50 80 D 0DC
ST 4,372 (0,3)
00062C 41 40 D 0AC
ST 4,372 (0,3)
000630 50 80 D 0DD
ST 4,372 (0,3)
000634 41 40 D 0DO
ST 4,372 (0,3)
000638 50 80 D 0DC
ST 4,372 (0,3)
00063C 50 80 D 0DD
ST 4,372 (0,3)
000640 50 80 D 0DC
ST 4,372 (0,3)

Appendix F: Programming Example 241
**PL/I OPTIMIZING COMPILER**

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>000644</td>
<td>50 F0 D 0D8</td>
<td></td>
</tr>
<tr>
<td>000648</td>
<td>41 40 D 0D8</td>
<td></td>
</tr>
<tr>
<td>000650</td>
<td>50 D0 D 0E4</td>
<td></td>
</tr>
<tr>
<td>000654</td>
<td>58 F0 3 02C</td>
<td></td>
</tr>
<tr>
<td>000658</td>
<td>50 F0 D 0EO</td>
<td></td>
</tr>
<tr>
<td>000660</td>
<td>50 40 3 194</td>
<td></td>
</tr>
<tr>
<td>000664</td>
<td>96 80 3 194</td>
<td></td>
</tr>
<tr>
<td>000668</td>
<td>18 55</td>
<td></td>
</tr>
<tr>
<td>00066A</td>
<td>41 10 3 180</td>
<td></td>
</tr>
<tr>
<td>00066E</td>
<td>58 F0 3 198</td>
<td></td>
</tr>
<tr>
<td>000720</td>
<td>05 EF</td>
<td></td>
</tr>
</tbody>
</table>

*STATEMENT NUMBER 31*

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>000674</td>
<td>D2 03 D 0C0 3 1FC</td>
<td></td>
</tr>
<tr>
<td>000677</td>
<td>D2 07 D 0C4 3 0E0</td>
<td></td>
</tr>
<tr>
<td>000680</td>
<td>50 80 0 DC4</td>
<td></td>
</tr>
<tr>
<td>00068C</td>
<td>50 40 3 19C</td>
<td></td>
</tr>
<tr>
<td>000690</td>
<td>D2 03 D 0CC 3 200</td>
<td></td>
</tr>
<tr>
<td>000696</td>
<td>D2 07 D 0DO 0 080</td>
<td></td>
</tr>
<tr>
<td>0006A0</td>
<td>50 00 D 0DO</td>
<td></td>
</tr>
<tr>
<td>0006A4</td>
<td>41 40 D 0DO</td>
<td></td>
</tr>
<tr>
<td>0006A8</td>
<td>50 40 3 1A0</td>
<td></td>
</tr>
<tr>
<td>0006AC</td>
<td>D2 01 D 0DA 3 204</td>
<td></td>
</tr>
<tr>
<td>0006B2</td>
<td>D2 07 D 0DC 3 0F8</td>
<td></td>
</tr>
<tr>
<td>0006BA</td>
<td>50 80 D 0DA</td>
<td></td>
</tr>
<tr>
<td>0006BC</td>
<td>50 80 D 0DC</td>
<td></td>
</tr>
<tr>
<td>0006C0</td>
<td>41 40 D 0DC</td>
<td></td>
</tr>
<tr>
<td>0006C4</td>
<td>50 40 3 1A4</td>
<td></td>
</tr>
<tr>
<td>0006C8</td>
<td>41 40 6 0CC</td>
<td></td>
</tr>
<tr>
<td>0006CC</td>
<td>50 80 3 1A8</td>
<td></td>
</tr>
<tr>
<td>0006D0</td>
<td>96 80 3 1A8</td>
<td></td>
</tr>
<tr>
<td>0006D4</td>
<td>18 55</td>
<td></td>
</tr>
<tr>
<td>0006D6</td>
<td>41 10 3 19C</td>
<td></td>
</tr>
<tr>
<td>0006DA</td>
<td>58 F0 3 1AC</td>
<td></td>
</tr>
<tr>
<td>0006DE</td>
<td>05 EF</td>
<td></td>
</tr>
</tbody>
</table>

*STATEMENT NUMBER 32*

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0006E0</td>
<td>D2 05 D 0C2 3 206</td>
<td></td>
</tr>
<tr>
<td>0006E6</td>
<td>D2 07 D 0C8 3 0DO</td>
<td></td>
</tr>
<tr>
<td>0006F0</td>
<td>50 E0 D 0C8</td>
<td></td>
</tr>
<tr>
<td>0006F4</td>
<td>41 40 D 0C8</td>
<td></td>
</tr>
<tr>
<td>0006F8</td>
<td>50 40 3 19C</td>
<td></td>
</tr>
<tr>
<td>0006FC</td>
<td>D2 07 D 0DO 3 100</td>
<td></td>
</tr>
<tr>
<td>000702</td>
<td>41 E0 D 0DO</td>
<td></td>
</tr>
</tbody>
</table>

*STATEMENT NUMBER 33*

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>000706</td>
<td>50 E0 D 0DO</td>
<td></td>
</tr>
<tr>
<td>00070A</td>
<td>41 40 D 0DO</td>
<td></td>
</tr>
<tr>
<td>000712</td>
<td>D2 01 D 0DA 3 204</td>
<td></td>
</tr>
<tr>
<td>000718</td>
<td>D2 07 D 0DC 3 0F8</td>
<td></td>
</tr>
<tr>
<td>00071E</td>
<td>41 80 D 0DA</td>
<td></td>
</tr>
<tr>
<td>000722</td>
<td>50 80 D 0DC</td>
<td></td>
</tr>
<tr>
<td>000726</td>
<td>41 40 D 0DC</td>
<td></td>
</tr>
<tr>
<td>000728</td>
<td>50 40 3 1A4</td>
<td></td>
</tr>
<tr>
<td>00072E</td>
<td>41 40 6 0CC</td>
<td></td>
</tr>
<tr>
<td>000732</td>
<td>50 40 3 1A8</td>
<td></td>
</tr>
<tr>
<td>000736</td>
<td>96 80 3 1A8</td>
<td></td>
</tr>
<tr>
<td>00073C</td>
<td>41 10 3 19C</td>
<td></td>
</tr>
<tr>
<td>000740</td>
<td>58 F0 3 1AC</td>
<td></td>
</tr>
<tr>
<td>000744</td>
<td>05 EF</td>
<td></td>
</tr>
</tbody>
</table>

*STATEMENT NUMBER 34*

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>000750</td>
<td>1B 11</td>
<td></td>
</tr>
<tr>
<td>000752</td>
<td>1B 55</td>
<td></td>
</tr>
<tr>
<td>000754</td>
<td>58 F0 3 1B4</td>
<td></td>
</tr>
<tr>
<td>000758</td>
<td>05 EF</td>
<td></td>
</tr>
</tbody>
</table>

*STATEMENT NUMBER 35*

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>00075A</td>
<td>41 40 6 D 0CO</td>
<td></td>
</tr>
<tr>
<td>00075E</td>
<td>50 40 3 1B8</td>
<td></td>
</tr>
<tr>
<td>000762</td>
<td>96 80 3 1B8</td>
<td></td>
</tr>
<tr>
<td>000764</td>
<td>424(3),X'80'</td>
<td></td>
</tr>
<tr>
<td>000768</td>
<td>41 10 3 1B8</td>
<td></td>
</tr>
<tr>
<td>00076C</td>
<td>58 F0 3 138</td>
<td></td>
</tr>
<tr>
<td>000770</td>
<td>05 EF</td>
<td></td>
</tr>
</tbody>
</table>

*STATEMENT NUMBER 36*

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>000772</td>
<td>18 0D</td>
<td></td>
</tr>
<tr>
<td>000774</td>
<td>58 D0 D 004</td>
<td></td>
</tr>
<tr>
<td>000778</td>
<td>58 E0 D 00C</td>
<td></td>
</tr>
<tr>
<td>00077C</td>
<td>98 2C D 01C</td>
<td></td>
</tr>
<tr>
<td>000780</td>
<td>05 1E</td>
<td></td>
</tr>
</tbody>
</table>

*STATEMENT NUMBER 41*

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>000782</td>
<td>07 07</td>
<td></td>
</tr>
</tbody>
</table>

*NOPR 7*
* REAL ENTRY
000784 90 EC D 00C
000787 00000000

* PROCEDURE
A

* REAL ENTRY
000788 90 EC D 00C
00078C 47 F0 F 014

* PROCEDURE BASE
BALR 2,0

* STATEMENT NUMBER 36
000784 DC C' A'
000787 DC AL1(1)

* STATEMENT NUMBER 37
0007DE 18 0D
0007E0 58 D0 D 004
0007E4 58 E0 D 00C
0007E8 98 2C D 01C
0007EC 05 1E

* STATEMENT NUMBER 38
0007F0 DC C' B'
0007F3 DC AL1(1)

* END PROGRAM
PL/I OPTIMIZING COMPILER

COMPILER DIAGNOSTIC MESSAGES

<table>
<thead>
<tr>
<th>ERROR ID</th>
<th>STMT</th>
<th>MESSAGE DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>IEL0413I</td>
<td>23</td>
<td>DECLARATION OF INTERNAL ENTRY NOT ALLOWED. DECLARATION OF 'A' IGNORED.</td>
</tr>
</tbody>
</table>

WARNING DIAGNOSTIC MESSAGES

<table>
<thead>
<tr>
<th>ERROR ID</th>
<th>STMT</th>
<th>MESSAGE DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>IEL0892I</td>
<td>6</td>
<td>TARGET STRING SHORTER THAN SOURCE. RESULT TRUNCATED ON ASSIGNMENT.</td>
</tr>
<tr>
<td>IEL0518I</td>
<td>20</td>
<td>'ONCOUNT' IS THE NAME OF A BUILTIN FUNCTION BUT ITS IMPLICIT DECLARATION DOES NOT IMPLY 'BUILTIN'.</td>
</tr>
<tr>
<td>IEL0916I</td>
<td>22</td>
<td>ITEM(S) 'SIZE' MAY BE UNINITIALIZED WHEN USED IN THIS BLOCK.</td>
</tr>
</tbody>
</table>

COMPILER INFORMATIVE MESSAGES

<table>
<thead>
<tr>
<th>ERROR ID</th>
<th>STMT</th>
<th>MESSAGE DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>IEL0541I</td>
<td>1, 9, 22, 36, 38</td>
<td>'ORDER' OPTION APPLIES TO THIS BLOCK. OPTIMIZATION MAY BE INHIBITED.</td>
</tr>
</tbody>
</table>

END OF COMPILER DIAGNOSTIC MESSAGES

<table>
<thead>
<tr>
<th>COMPIL TIME</th>
<th>SPILL FILE:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.12 MIN</td>
<td>2 RECORDS, SIZE 3891</td>
</tr>
</tbody>
</table>

Diagnostic messages and an end of compile step message generated by the compiler. All diagnostic messages generated by the optimizing compiler are documented in the publication OS Optimizing Compiler: Messages.

1. "ERROR ID" This identifies the message as originating from the optimizing compiler (IEL), and gives the message number.
2. "L" This is the severity level of the message.
3. "STMT" This gives the number of the statement in which the error occurred.
4. Compile time in minutes. This time includes the preprocessor.
5. This gives the number of records "spilled" into auxiliary storage and the size in bytes of the spill file.
The page is a printed document that appears to be a listing from a linkage editor. It includes various entries and references, possibly related to programming or system configuration. The text suggests that it is part of a larger document, possibly a programming example or guide. The page contains a cross-reference table and details on control sections and entries. There are annotations on the page, indicating different sections and notes about the content. The annotations are written in a way that they are aligned with the corresponding entries or notes on the page, suggesting a method of documentation or instruction.
<table>
<thead>
<tr>
<th>NAME</th>
<th>ORIGIN</th>
<th>LENGTH</th>
<th>NAME</th>
<th>LOCATION</th>
<th>NAME</th>
<th>LOCATION</th>
<th>NAME</th>
<th>LOCATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBMERR1</td>
<td>3120</td>
<td>60</td>
<td>IBMERRA</td>
<td>3120</td>
<td>IBMERRB</td>
<td>3162</td>
<td>IBMERRC</td>
<td>3734</td>
</tr>
<tr>
<td>IBMJD1</td>
<td>3720</td>
<td>6</td>
<td>IBMJD1A</td>
<td>3700</td>
<td>IBMJD1B</td>
<td>3702</td>
<td>IBMJD1C</td>
<td>3704</td>
</tr>
<tr>
<td>IBMJTT1</td>
<td>3870</td>
<td>6</td>
<td>IBMJTTA</td>
<td>3700</td>
<td>IBMJTTB</td>
<td>3702</td>
<td>IBMJTTC</td>
<td>3704</td>
</tr>
<tr>
<td>IBMBO1</td>
<td>3880</td>
<td>10</td>
<td>IBMBO1A</td>
<td>3800</td>
<td>IBMBO1B</td>
<td>3802</td>
<td>IBMBO1C</td>
<td>3804</td>
</tr>
<tr>
<td>IBMBSR1</td>
<td>3980</td>
<td>20</td>
<td>IBMBSR1A</td>
<td>3900</td>
<td>IBMBSR1B</td>
<td>3902</td>
<td>IBMBSR1C</td>
<td>3904</td>
</tr>
<tr>
<td>IBMBJ1</td>
<td>4180</td>
<td>20</td>
<td>IBMBJ1A</td>
<td>4100</td>
<td>IBMBJ1B</td>
<td>4102</td>
<td>IBMBJ1C</td>
<td>4104</td>
</tr>
<tr>
<td>IBMBSP1</td>
<td>4400</td>
<td>120</td>
<td>IBMBSP1A</td>
<td>4400</td>
<td>IBMBSP1B</td>
<td>4402</td>
<td>IBMBSP1C</td>
<td>4404</td>
</tr>
<tr>
<td>IBMBCE1</td>
<td>4620</td>
<td>8</td>
<td>IBMBCE1A</td>
<td>4600</td>
<td>IBMBCE1B</td>
<td>4602</td>
<td>IBMBCE1C</td>
<td>4604</td>
</tr>
<tr>
<td>IBMB.req</td>
<td>4770</td>
<td>4</td>
<td>IBMB.reqA</td>
<td>4700</td>
<td>IBMB.reqB</td>
<td>4702</td>
<td>IBMB.reqC</td>
<td>4704</td>
</tr>
<tr>
<td>IBMBSCV</td>
<td>4780</td>
<td>240</td>
<td>IBMBSCV1A</td>
<td>4700</td>
<td>IBMBSCV1B</td>
<td>4702</td>
<td>IBMBSCV1C</td>
<td>4704</td>
</tr>
</tbody>
</table>

LOCATION REFERS TO SYMBOL IN CONTROL SECTION

<table>
<thead>
<tr>
<th>LOCATION</th>
<th>REFERS TO SYMBOL</th>
<th>IN CONTROL SECTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>PLIMAIN</td>
<td>PLIMAIN</td>
</tr>
<tr>
<td>18</td>
<td>PLIFLOW</td>
<td>$UNRESOLVED(W)</td>
</tr>
<tr>
<td>26</td>
<td>PLICOUNT</td>
<td>$UNRESOLVED(W)</td>
</tr>
<tr>
<td>38</td>
<td>IBMFIR</td>
<td>IBMFIR1</td>
</tr>
<tr>
<td>40</td>
<td>IBMFIR1</td>
<td>IBMFIR1</td>
</tr>
<tr>
<td>74</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>76</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>84</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>94</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>96</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>98</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>100</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>102</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>104</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>106</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>108</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>110</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>112</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>114</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>116</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>118</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>120</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>122</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>124</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>126</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>128</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>130</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>132</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>134</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>136</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>138</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>140</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>142</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>144</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>146</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>148</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>150</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>152</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>154</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>156</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>158</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>160</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>162</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>164</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>166</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>168</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>170</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>172</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>174</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>176</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>178</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>180</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>182</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>184</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>186</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>188</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>190</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
</tbody>
</table>

LOCATION REFERS TO SYMBOL IN CONTROL SECTION

<table>
<thead>
<tr>
<th>LOCATION</th>
<th>REFERS TO SYMBOL</th>
<th>IN CONTROL SECTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>PLIMAIN</td>
<td>PLIMAIN</td>
</tr>
<tr>
<td>18</td>
<td>PLIFLOW</td>
<td>$UNRESOLVED(W)</td>
</tr>
<tr>
<td>26</td>
<td>PLICOUNT</td>
<td>$UNRESOLVED(W)</td>
</tr>
<tr>
<td>38</td>
<td>IBMFIR</td>
<td>IBMFIR1</td>
</tr>
<tr>
<td>40</td>
<td>IBMFIR1</td>
<td>IBMFIR1</td>
</tr>
<tr>
<td>74</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>76</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>84</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>94</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>96</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>98</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>100</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>102</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>104</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>106</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>108</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>110</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>112</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>114</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>116</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>118</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>120</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>122</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>124</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>126</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>128</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>130</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>132</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>134</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>136</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>138</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>140</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>142</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>144</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>146</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>148</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>150</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>152</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>154</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>156</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>158</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>160</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>162</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>164</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>166</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>168</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>170</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>172</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>174</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>176</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>178</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>180</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>182</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>184</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>186</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>188</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>190</td>
<td>*SAMPLE1</td>
<td>*SAMPLE1</td>
</tr>
<tr>
<td>LOCATION</td>
<td>SYSPINT</td>
<td>SYSPINT</td>
</tr>
<tr>
<td>----------</td>
<td>---------</td>
<td>---------</td>
</tr>
<tr>
<td>19C</td>
<td>SYSINT</td>
<td>SYSINT</td>
</tr>
<tr>
<td>1AC</td>
<td>IBMKPROMA</td>
<td>IBMKPROM1</td>
</tr>
<tr>
<td>188</td>
<td>IBMKPROMA</td>
<td>IBMKPROM1</td>
</tr>
<tr>
<td>208</td>
<td>IBMKPROMA</td>
<td>IBMKPROM1</td>
</tr>
<tr>
<td>220</td>
<td>IBMKPROMA</td>
<td>IBMKPROM1</td>
</tr>
<tr>
<td>284</td>
<td>IBMKPROMA</td>
<td>IBMKPROM1</td>
</tr>
<tr>
<td>304</td>
<td>IBMKPROMA</td>
<td>IBMKPROM1</td>
</tr>
<tr>
<td>34C</td>
<td>IBMKPROMA</td>
<td>IBMKPROM1</td>
</tr>
<tr>
<td>468</td>
<td>IBMKPROMA</td>
<td>IBMKPROM1</td>
</tr>
<tr>
<td>480</td>
<td>IBMKPROMA</td>
<td>IBMKPROM1</td>
</tr>
<tr>
<td>640</td>
<td>IBMKPROMA</td>
<td>IBMKPROM1</td>
</tr>
<tr>
<td>800</td>
<td>IBMKPROMA</td>
<td>IBMKPROM1</td>
</tr>
<tr>
<td>15B0</td>
<td>IBMKPEA</td>
<td>$UNRESOLVED(W)</td>
</tr>
<tr>
<td>1840</td>
<td>IBMKPEA</td>
<td>$UNRESOLVED(W)</td>
</tr>
<tr>
<td>1848</td>
<td>IBMKPEA</td>
<td>$UNRESOLVED(W)</td>
</tr>
<tr>
<td>1800</td>
<td>IBMKPEA</td>
<td>$UNRESOLVED(W)</td>
</tr>
<tr>
<td>1830</td>
<td>IBMKPEA</td>
<td>$UNRESOLVED(W)</td>
</tr>
<tr>
<td>183C</td>
<td>IBMKPEA</td>
<td>$UNRESOLVED(W)</td>
</tr>
<tr>
<td>1868</td>
<td>IBMKPEA</td>
<td>$UNRESOLVED(W)</td>
</tr>
<tr>
<td>1834</td>
<td>IBMKPEA</td>
<td>$UNRESOLVED(W)</td>
</tr>
<tr>
<td>195C</td>
<td>IBMKPEA</td>
<td>$UNRESOLVED(W)</td>
</tr>
<tr>
<td>196C</td>
<td>IBMKPEA</td>
<td>$UNRESOLVED(W)</td>
</tr>
<tr>
<td>1974</td>
<td>IBMKPEA</td>
<td>$UNRESOLVED(W)</td>
</tr>
<tr>
<td>198C</td>
<td>IBMKPEA</td>
<td>$UNRESOLVED(W)</td>
</tr>
<tr>
<td>1990</td>
<td>IBMKPEA</td>
<td>$UNRESOLVED(W)</td>
</tr>
<tr>
<td>1998</td>
<td>IBMKPEA</td>
<td>$UNRESOLVED(W)</td>
</tr>
<tr>
<td>19C8</td>
<td>IBMKPEA</td>
<td>$UNRESOLVED(W)</td>
</tr>
<tr>
<td>1A1C</td>
<td>IBMKPEA</td>
<td>$UNRESOLVED(W)</td>
</tr>
<tr>
<td>1A0C</td>
<td>IBMKPEA</td>
<td>$UNRESOLVED(W)</td>
</tr>
<tr>
<td>199C</td>
<td>IBMKPEA</td>
<td>$UNRESOLVED(W)</td>
</tr>
<tr>
<td>19D4</td>
<td>IBMKPEA</td>
<td>$UNRESOLVED(W)</td>
</tr>
<tr>
<td>1984</td>
<td>IBMKPEA</td>
<td>$UNRESOLVED(W)</td>
</tr>
<tr>
<td>198D</td>
<td>IBMKPEA</td>
<td>$UNRESOLVED(W)</td>
</tr>
<tr>
<td>19B8</td>
<td>IBMKPEA</td>
<td>$UNRESOLVED(W)</td>
</tr>
<tr>
<td>19A8</td>
<td>IBMKPEA</td>
<td>$UNRESOLVED(W)</td>
</tr>
<tr>
<td>198A</td>
<td>IBMKPEA</td>
<td>$UNRESOLVED(W)</td>
</tr>
<tr>
<td>198B</td>
<td>IBMKPEA</td>
<td>$UNRESOLVED(W)</td>
</tr>
<tr>
<td>1998</td>
<td>IBMKPEA</td>
<td>$UNRESOLVED(W)</td>
</tr>
<tr>
<td>1A08</td>
<td>IBMKPEA</td>
<td>$UNRESOLVED(W)</td>
</tr>
<tr>
<td>1A08</td>
<td>IBMKPEA</td>
<td>$UNRESOLVED(W)</td>
</tr>
<tr>
<td>1968</td>
<td>IBMKPEA</td>
<td>$UNRESOLVED(W)</td>
</tr>
<tr>
<td>19C4</td>
<td>IBMKPEA</td>
<td>$UNRESOLVED(W)</td>
</tr>
<tr>
<td>1A2C</td>
<td>IBMKPEA</td>
<td>$UNRESOLVED(W)</td>
</tr>
<tr>
<td>1A3C</td>
<td>IBMKPEA</td>
<td>$UNRESOLVED(W)</td>
</tr>
<tr>
<td>1A4C</td>
<td>IBMKPEA</td>
<td>$UNRESOLVED(W)</td>
</tr>
<tr>
<td>2054</td>
<td>IBMKPEA</td>
<td>$UNRESOLVED(W)</td>
</tr>
<tr>
<td>2064</td>
<td>IBMKPEA</td>
<td>$UNRESOLVED(W)</td>
</tr>
<tr>
<td>205C</td>
<td>IBMKPEA</td>
<td>$UNRESOLVED(W)</td>
</tr>
</tbody>
</table>
## LOCATION REFERS TO SYMBOL IN CONTROL SECTION

<table>
<thead>
<tr>
<th>Location</th>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>22F4</td>
<td>IBMBCFX</td>
<td>$UNRESOLVED(W)</td>
</tr>
<tr>
<td>2304</td>
<td>IBMBCHPP</td>
<td>IBMCH01</td>
</tr>
<tr>
<td>25EC</td>
<td>IBMBCSCA</td>
<td>IBMBCV1</td>
</tr>
<tr>
<td>2C9C</td>
<td>IBMBSAOA</td>
<td>$UNRESOLVED(W)</td>
</tr>
<tr>
<td>2C94</td>
<td>IBMBSPOA</td>
<td>IBMSP01</td>
</tr>
<tr>
<td>2C98</td>
<td>IBMBSFOA</td>
<td>IBMBSPO1</td>
</tr>
<tr>
<td>2C80</td>
<td>IBMBSIST</td>
<td>$UNRESOLVED(W)</td>
</tr>
<tr>
<td>2F00</td>
<td>IBMBSPLB</td>
<td>IBMBSPL1</td>
</tr>
<tr>
<td>2EF8</td>
<td>IBMBCLLA</td>
<td>IBMBCCL1</td>
</tr>
<tr>
<td>37B0</td>
<td>IBMBERCA</td>
<td>$UNRESOLVED(W)</td>
</tr>
<tr>
<td>39CC</td>
<td>IBMBCICB</td>
<td>IBMBCICB</td>
</tr>
<tr>
<td>41AC</td>
<td>IBMBCACA</td>
<td>$UNRESOLVED(W)</td>
</tr>
<tr>
<td>41A4</td>
<td>IBMBC2CA</td>
<td>$UNRESOLVED(W)</td>
</tr>
<tr>
<td>419C</td>
<td>IBMBSIST</td>
<td>$UNRESOLVED(W)</td>
</tr>
<tr>
<td>4564</td>
<td>IBMBCDP</td>
<td>IBMBC01</td>
</tr>
<tr>
<td>4574</td>
<td>IBMBCMPP</td>
<td>$UNRESOLVED(W)</td>
</tr>
<tr>
<td>4580</td>
<td>IBMBCHEX</td>
<td>IBMCH01</td>
</tr>
<tr>
<td>4588</td>
<td>IBMBCHEF</td>
<td>IBMCH01</td>
</tr>
<tr>
<td>459C</td>
<td>IBMBCMPE</td>
<td>$UNRESOLVED(W)</td>
</tr>
<tr>
<td>4590</td>
<td>IBMBCPBE</td>
<td>$UNRESOLVED(W)</td>
</tr>
</tbody>
</table>

## LOCATION 20 REQUESTS CUMULATIVE PSEUDO REGISTER LENGTH

<table>
<thead>
<tr>
<th>Entry Address</th>
<th>Cumulative Pseudo Register Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>

**TOTAL LENGTH** 49C0

****GO**  **DOES NOT EXIST BUT HAS BEEN ADDED TO DATA SET**

---

**SAMPLE PROGRAM**

DATE = 73/04/13, TIME = 21.18.17

END SAMPLE PROGRAM
Appendix G: Running under a Virtual Storage Operating System (OS/VS)

OS/VS1 and OS/VS2 are the virtual-storage equivalents of OS/MFT and OS/MVT respectively. In general, a program that compiles and executes successfully under OS/MFT or OS/MVT will do so under OS/VS.

OS/VS has the advantage that the size of the partition or region used by the program is not limited by the amount of real storage available. In the case of PL/I programs, partitions or regions large enough to give maximum efficiency during both compilation and execution can be used, and the segmentation of programs to fit restricted amounts of storage becomes unnecessary.

Virtual partitions or regions can be allocated in multiples of 64K bytes. The amount of real storage available to a program at any time is under the control of the Operating System, and cannot be specified by the programmer. The storage requirements given below refer to the amounts of virtual storage available.

The compiler will run in the minimum virtual partition or region size of 64K bytes. However, increased efficiency can be obtained by using a partition or region size large enough to prevent the compiler from using its spill file. The use of partition or region sizes in excess of those required to prevent the compiler from using its spill file may cause a slight degradation in performance, but this will not usually be significant unless the amount of real storage allocated by the operating system is small.

It is inadvisable to limit the amount of storage available to the compiler by specifying the SIZE option. This option should be used only if you require to reserve space for other routines, such as ones that invoke the compiler dynamically. In any case, the compiler requires at least 52K bytes of storage under VS1 and at least 54K bytes under VS2.

The sizes of records on the compiler spill file depend on the amount of storage available to the compiler. Figure G-1 gives record sizes under VS1. For VS2, the specified storage size limits should be increased by 2K bytes.

<table>
<thead>
<tr>
<th>Storage (bytes)</th>
<th>Record size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>52-62K</td>
<td>1091</td>
</tr>
<tr>
<td>62-78K</td>
<td>1691</td>
</tr>
<tr>
<td>78-84K</td>
<td>3491</td>
</tr>
<tr>
<td>Over 84K</td>
<td>4051</td>
</tr>
</tbody>
</table>

Figure G-1. Compiler spill file record sizes

Note that the 2311 Disk Storage Unit is not supported by OS/VS.
Where more than one page reference is given, the major reference is first.

A page reference to a paragraph split between two pages will refer to the first of the two pages.

$LKLBDSN$ parameter in cataloged procedure 150

* parameter of DD statement 11
* PROCESS statement 5

%INCLUDE statement 44
%PAGE statement 26,35
%SKIP statement 26,35

abbreviated form of compiler options 21
absolute addresses 49,70
access method services 133
access methods 80,131
access speed
improving, for INDEXED data sets 110
improving, for REGIONAL data sets 118
accessing a VSAM data set
entry sequenced data set 137
key sequenced data set 136
addressing 49
advanced checkpoint/restart 189
aggregate length table 37
AGGREGATE option 22
ALIAS statement (linkage editor) 58
aliases 58
American National Standard control characters (see ANS)
American Standard Code for Information Interchange (see ASCII)
AMP parameter 137
ANS (American National Standard) control characters
printers 95,105
punched card devices 105
source listing 26
specifying in JCL 197
APAR (Authorized Program Analysis Report) 207
areas 199
argument passed to main procedure 31
arguments in checkpoint/restart 200
arrays
length table 37
mapping 17
maximum number of dimensions 199
ASCII (American Standard Code for Information Interchange) 74,75,194
for paper tape, specifying 194
option of ENVIRONMENT attribute 74
records 76
Assembler language linkage 165
abnormal termination 169,173
Assembler-PL/I-Assembler 173
calling Assembler routines from PL/I 169

Assembler language linkage (continued)
calling PL/I procedures from Assembler 170
error handling 173
establishing PL/I environment 165
invoking PL/I procedure 165
linkage conventions 165
no main PL/I procedure 170
use of register 12 169

Assembler language listing 40
ASSEMBLER option 174
asterisk (*) parameter of DD statement 11
ATTACH macro instruction 45
attribute listing 22,29,36
ATTRIBUTES option 22
automatic library call
DD statement for 51
introduction 50
main discussion 52
suppressing 54,69
use of by loader 64,69
use of by programmer 141
auxiliary storage (see storage)

base library (SYS1.PLEBASE) 53,150
basic access technique 81
Basic Direct Access Method (BDAM) 81
Basic Indexed Sequential Access Method (BISAM) 81
Basic Sequential Access Method (BSAM) 81
batched compilation 41,62
BCD (Binary Coded Decimal) compiler options 22
magnetic tape translation 88,197
BDAM (Basic Direct Access Method) 81
Binary Coded Decimal (see BCD)
BISAM (Basic Indexed Sequential Access Method) 81
blanks, removal of 17
BLKSIZE option of ENVIRONMENT attribute 74
BLKSIZE subparameter of DCB parameter 194,74
block size 194
CONSECUTIVE data sets
record I/O 101,103
stream I/O 91,92
INDEXED data sets 110
introduction 9
PRINT files 95
REGIONAL data sets 118
specifying 74,194
system output device (SYSOUT) 13
blocking (in general) 74
boundary alignment 17
branching, trace table showing 160
BSAM (Basic Sequential Access Method) 81
buffers
contents, in dump 161
default storage allocations 18
general discussion 80
specifying number of 194
BUFFERS option of ENVIRONMENT 196
BUFFNO subparameter of DCB parameter 194
BUFPOFF option 74
built-in functions 199
recognized by context 199
without arguments 199
BUILTIN attribute 199
Burroughs code for paper tape, specifying 194
bypassing errors 160
CALL macro instruction 45
CALL option (loader) 69
capacity record 118
card devices (see punched card devices)
card output (see punched card output)
cataloged data sets 73
cataloged procedures 149
creating new 152
DD statements 7
definition 2
IBM-supplied 152
input data set 152
invoking 149
modifying 151
multitasking 150
region size 152
shared library 213
standard files 13
with MFT 152
with MVT 152
CATLG subparameter of DISP parameter 73
chained scheduling 80, 196
channel programs, specifying number 195
character set specification 22
character, invalid 196
CHARSET option 22
CHECK condition 160
CHECK prefix, use of 160
checkout compiler modules 63
checkout, program (continued)
system failure 160
trace information 162
unidentified errors 159
use of a standard set of checkout statements 161
checkpoint/restart 189, 200
COBOL structures in aggregate length table 37
code identifying object module 24
code subparameter of DCB parameter 194
column binary mode for card device, specifying 195
combining procedures 54
comments, removal of 17
common areas 38, 48
compatibility interface, ISAM/VSAM 137
compatibility with the PL/I (F) compiler 199
compilation
batched 41, 62
speed of 28
suppressing 158
COMPILE option 23
compile-time processing (see preprocessing)
compiler
failure 160, 207
failure, suspected 158
general description 15
compiler options 21
abbreviations 22
batched compilation 41
continuation line for 21
defaults 22
descriptions 22
installation deletions 24
introduction 15
preprocessor 43
specifying 21
summary table 22
use in checking out program 157, 158
used for compiler listings 34
completion codes (see return codes)
concatenating data sets 78
concatenating libraries 142
condition built-in function values in trace 162
condition handling 161
Models 91 and 195 211
conditional compilation 23
conditional execution of job step 163
conditional subparameter of DISP 79
conditions 200
CONSECUTIVE data sets
accessing in record I/O 102
accessing in stream I/O 92
creating in record I/O 101
creating in stream I/O 91
general description 77
introduction 7
continuation line for compiler options 21
control area 131
control characters
card devices 105
printers 95, 106
specifying in JCL 197, 81
CONTROL option 23
control interval 131
CREATING in record I/O 101
creating in stream I/O 91
control program (see operating system)
control sections
  identification 38
  length 38
  listing, linkage editor 56
  listing, loader 69
control statements, linkage editor 57
listing of 54
control variables in DO statement 200
conversational processing (see TSO)
conversion feature of 2400-series tape drives 197
COPY option, use of 159
COUNT option 24
  execution-time 33
  restriction 62
cross-reference listing 22,30
compiler 36
linkage editor 57
cylinders
  definition 89
  index 108
overflow area 112,108
specifying overflow area 194,196
CYLOFL subparameter of DCB parameter 194
data sets (continued)
  DD statements 19
ddnames 78,19
dedicated 20
defining 78
  for record files 101
  for stream files 91
definition of term 73
device class 10
device type 8,10
direct 77
disposition 11
dissociating from PL/I file 83
generation data group 74
independent overflow area 114,108
index area 108,114
INDEXED (see INDEXED data sets)
  indexed sequential 77
  indexes 107
input 19
input, and cataloged procedures 152
labels
copying from 193
general description 77
  in library data sets 141
  modification by data management 83
nonstandard 77,103
limiting search extent 195
linkage editor 51
listings 20
loader 66
magnetic tape 101
master index 108,114
members 77
messages 20
names
  introduction 9,10
  main discussion 73,78
organization 77
output 19
overflow area 108,112
partitioned (see libraries)
prime data area 108,114
printer line spacing 79
qualified names 73,74
re-creation 115
record formats 75
  specifying in JCL 195,196
record type 9,10
records 74
region numbers 116
REGIONAL (see REGIONAL data sets)
reorganizing INDEXED data sets 115
retrieval of cataloged data sets 73
sequential 77
sort/merge 178
source program 19
source statement library 20
storage for (see storage)
telecommunications 77
temporary 20,9
track index 107,108
unlabeled 77
unnamed 74,10
updating CONSECUTIVE data sets 102
updating INDEXED data sets 114
updating REGIONAL data sets 120
use of DCB subparameters 193
data sets (continued)

volume serial number 9,10
7-track tape 197
data, invalid 159
DB-format records 75,76
DCB (data control block) 81
DCB parameter
(see also DCB subparameters)
introduction 9,10
main discussion 79
summary appendix 193
DCB subparameters
BLKSIZE 194
BUFNO 194,195
CODE 194
CYLOFL 194
DEN 194
DSORG 195
for CONSECUTIVE data sets 103
for INDEXED data sets 110
for REGIONAL data sets 120
KEYLEN 195
LIMCT 195
LRECL 195
MODE 195
NCM 195
NTM 196
OPTC'D 196
overriding in cataloged procedures 152
RECFM 196
RKP 197
STACK 197
TRTCH 197
DD (data definition) statements 78
adding, to cataloged procedures 151
creating a library 142
ddnames (see ddnames)
essential parameters 10
for checkpoint/restart data sets 190
for INDEXED data sets 110,114
for input data set in cataloged
procedures 153
for linkage editor data sets 51
for loader data sets 66
for record I/O 101
for sort/merge data sets 178
for standard data sets 19
for stream I/O 91
for VSAM data sets 137
introduction 2,8
modifying, in cataloged procedures 151
parameters 8
PLIDUMP 162
ddnames 78
definition of term 8
for checkpoint/restart data sets 190
for linkage editor data sets 51
for loader data sets 66
for sort/merge data sets 180
for standard data sets 19
in dynamic invocation of compiler 45
deblocking of records 74,80
debugging (see checkout, program)
DECK option 24
DECLARE statement labels 204
dedicated data sets 20
dedicated workfiles 150
default options 20
DEFINED attribute 200
delimiter statement (job control
language) 11
demounting volumes, instructions for 79
DEN subparameter of DCB parameter 88,194
density, recording, magnetic tape 88,194
dependent declarations 200
depth of replacement maximum 43
device classes 10
for linkage editor data sets 52
for loader data sets 66
device description 78
device independence of source program 78
device specification 78,8,10
ddnames 79
diagnostic messages (see messages)
dictionary-build stage 17
direct data sets 77
direct-access devices
specifying storage requirements 9,89
specifying write validity check 196
directory, library 142,143
DISP parameter 42
introduction 80,10,11
main discussion 79
DISPLAY statement 200
DO statement control variables
for linkage editor 51
for loader 66
DSB (dynamic storage area)
trace 162
DSCB (data set control block) 77
for library 143
dsname parameter 78,9,10
DSORG subparameter of DCB parameter 195
dummy records
INDEXED data sets 114,196
REGIONAL data sets 118,120
dump option 24
dumps 161
dumps from PL/I programs 200
dynamic checkout facilities 160
dynamically loaded modules 59
EBCDIC (Extended Binary Coded Decimal
Interchange Code)
alternative codes 74
compiler option for source program 22
specifying mode for card devices 195
specifying translation to ECD 88,197
embedded keys 111,112
END instruction 49
END statement 37
ENDPAGE condition 201
ENTRY address 56
entry names 201
entry point listings
linkage editor 56
loader 70
entry sequenced data set 131
entry variables as source of error 159
ENVIRONMENT attribute 9,201
environment, PL/I, in Assembler language
linkage 165,170
EP option (loader) 69
error correction by compiler 157,201
error handling
Assembler-PL/I linkages 166,173
Models 91 and 195 211

254
error messages (see messages)
errors in program (see checkout, program)
(see also problem determination)
errors, operating 159
ESD (external symbol dictionary) 38
ESD option 24
ESSD (see entry sequenced data set)
exception (definition of term) 211
EXCLUSIVE attribute 201
exclusive calls 62
EXEC statements 19
continuation line 21
for linkage editor 51
for loader 66
locating load module 141
modifying, in cataloged
procedures 151,152
option list maximum length 21
PARM parameter 21
specifying compiler options 21
specifying execution-time options 31
executable load module labeling 54
executable program (definition) 15
execution
essential job control language 5
suppressing 158
execution-time options 21
descriptions 31
specifying 30
Extended Binary Coded Decimal Interchange
Code (see EBCDIC)
external references
definition 48
in ESD listing 38
in linkage editor listings 57
resolution by linkage editor
automatic library call 52
suppressing automatic library
call 54
unresolved 54,56
suppressing automatic library
call 54
external symbol dictionary (ESD) 48
F-format records 75
FB-format records 75
FS-format records 75
FCB (file control block) 163
FETCH statement 62
fetchable load modules 62
files
attributes 81
closing 83
information from PLIDUMP 163
introduction 7
opening 81
specifying number of channel
programs 195
standard 13
SYSIN 13
SYSPRINT 13
TRANSIENT 77
variable, as source of error 159
final-assembly stage 17
fix-ups for program product faults 209
fixed-length records 75,9
FLAG option 24
flow of control, tracing 160
FLOW option 24,160
execution-time 34
format descriptor card
optical mark read 85
read column eliminate 86
format of records (see record format)
formatted dump option 24
FORTRAN arrays in aggregate length
table 37
Friden code for paper tape, specifying 194
FS-format records 75
FUNC subparameter of DCB 195
generation data group 74
GET macro instruction 81
GONUMBER option 25
GOSTMT option 25
header label 77
heading information in listing 35
hexadecimal address representation in
ESD 39
hexadecimal dumps 161
I/O (see input/output)
IBG (interblock gap) 74
IBM code for paper tape, specifying 194
IBM control character, specifying 197
IBM programming support 207,160
IBM service aid program IMAPFLS 209
IBMBER 79
IBMTPIRA 39
IBMBSTAB 97
identifier listing 36
IELOAA 45
IEWL 51
IEWLDRGO 66
IHEDUMP 200
IMAPFLS service aid program 209
imprecise interrupts 211
IMPRECISE option 25,211
INCLUDE option 25
INCLUDE statement (linkage editor) 59
including source statements from a
library 44
independent overflow area 114,108
specifying in JCL 196
index (see INDEXED data sets)
index data set (VSAM) 131
index set (VSAM) 131
INDEXED data sets 108
adding records to 114,108,112
creation 108
deleted (dummy) records 114,196
index area 108,114
separate DD statement for 110
indexes 107
introduction 77
master index 114,196
overflow area 112,114,196
separate DD statement for 110,112
overflow records 194

Index 255
INDEXED data sets (continued)

prime area 114
separate DD statement for 110
specifying key position 197
specifying number of tracks per index 196
SYSOUT device restriction 110
INITIAL attribute 37
initial storage area (ISA) 31
initial volume label 77
initialization 15, 36, 39
input
compiler
data in the input stream 11, 98
in cataloged procedures 153
linkage editor 57
loader 66
input/output
(see also: data sets; input; output)
access methods 80
defining data sets for record files 101
defining data sets for stream files 91
device independence of source program 78
device specification 8
improving transmission time 196
introduction 7
locate mode 80
move mode 80
INSERT statement (linkage editor) 61
INSOURCE option 25
installation factors for cataloged procedures 152
interblock gap (IBG) 74
interlanguage communication between PL/I and Assembler 165
interrecord gap (see interblock gap)
interrupt (definition of term) 211
Assembler-PL/I linkages 166, 173
Models 91 and 195 211
ISA (initial storage area) 31
ISAM/VSAM compatibility interface 137
ISASIZE option 31

JCL (see job control language)
job (definition) 2
job control language (JCL) 41
checkpoint/restart 189
creating a library 142
DCB subparameters 193
for CONSECUTIVE data sets 103
for INDEXED data sets 110
for REGIONAL data sets 120
defining data set libraries 142
essential 5
for compilation 17, 20
for linkage editor 50, 53
for loader 64, 67
for sort/merge data sets 178
for VSAM data sets 137
introduction 2
JOB statement 2
MSGCLASS parameter 34
MSGLEVEL parameter 34
job step 2
JOBLIB DD statement 142
message processing programs 130
MPP (message processing program) 130
key sequenced data set 131
accessing 136
creation 133
keying records
INDEXED data sets 111, 112
REGIONAL data sets 118
specifying key length 195
specifying key position 197
KEYLEN subparameter of DCB parameter 195
keypunch (see punched card devices)
KSDS (see key sequenced data set)
LABEL parameter 78
label variables as source of error 159
labeling data sets 77
labeling volumes 73
LEAVE option 79
length of record, specifying 195, 74
LET option (linkage editor) 54
LET option (loader) 69
libraries 142
base library (SYS1.PLIBASE) 53
calling additional 58
creating 142
creating members 143
directory 143
including source statements from 44
multitasking library (SYS1.PLITASK) 53
structure 145
system procedure library (SYS1.PROCLIB) 141, 149
system program library (SYS1.LINKLIB) 141
types of 141
use by linkage editor 141
use by loader 141
use by operating system 142
use by PL/I program 142
library subroutines
control sections for 48
data set for 53
dynamic calling 49
ESD entries for 39
external reference resolution 56
failure of 160
failure of, suspected 158
in overlay structures 62
introduction 15
link-editing 49
multitasking version and cataloged procedures 150
using cataloged procedures 150
library, automatic call (see automatic library call)
LIMCT subparameter of DCB parameter 195
line numbers
and offsets, table of 28
in messages 25
in source listing 28
preprocessor 35
LINE option/format item 95
line size
default, and overriding it 96
specification 95
line spacing, printers 106, 95
specifying in JCL 197, 88
LINECOUNT option 25
LINESIZE option 98
LINK macro instruction 45
link-pack area 64, 69
linkage editor 47
(see also: load modules; loader)
ALIAS statement 58
checkout 158
choice of linkage editor or loader 47
control statement listing 55
control statements 57
cross-reference listing 57
data sets 51
DD statements 51
dnames 51
device classes 52
input 52, 58
job control language for 50
listings 55
messages 55
NAME statement 58, 27, 41
non-multitasking program 53
optional facilities 53
output 52
output and cataloged procedures 149
output to a library 141
overlapping (see overlaying)
region size 152
return code 0004 56
specifying storage for 54
storage requirements 50
suppressing automatic library call 54
suppressing link-editing 158
system program library (SYS1.LINKLIB) 141
temporary workspace 52
use by operating system 141
LINKEDIT (program alias) 51
LIST option 25
LIST option (linkage editor) 54
listings 22, 34
aggregate lengths 37
attribute 36
cataloged procedures 149
cross-reference 36
dumps 161
external symbol dictionary 38
general discussion 34
identifier 36
linkage editor 55
loader 70
nesting level in 36
object module 40
of compiler options 35
preprocessor input 35
preprocessor messages 35
sort/merge 181
source program 35
statement offset addresses 38
static internal control section 40
table of options 35
use in checking out program 157
with APARS 207
listings produced by programming
eexample 215
LMESSAGE option 26
load modules
ccontrol section listing 70
definition 3
disposition statement 56
libraries (see libraries)
location 141, 142
map option 56, 69
maximum size 52
name
compiler 27, 41
linkage editor 58
replacement 58
separation 58
structure 48
loader 64, 67
(see also: linkage editor; load modules)
choice of linkage editor or loader 47
data sets 66
DD statements for loader data sets 66
dnames 66
device classes 66
dumps 161
external reference resolution 69
general description 64
input 66
job control language for 64
listings 70
messages 70
module map 70
optional facilities 68
output 67
specifying entry point of program to 69
specifying storage for 70
storage requirements 60
LOAD (program alias) 66
locate mode input/output 80
locator variables as source of error 159
locked records 202
logical record 131
looping, preventing 159
LRECL subparameter of DCB parameter 195, 74
machine control characters (see control
characters)
machine errors 160
machine instruction listing 40
MACRO option 26
magnetic tapes
accessing, without standard
labels 93, 103
main discussion 88
special requirement for 7-track 197
specifying recording density 194
main procedure and Assembler language 170
MAP option
compiler 26
linkage editor 54, 56
loader 69
margin indicator option 26
MARGINI option 26
MARGINS option 26
master index (see INDEXED data sets)
MCP (message control program) 124
MDECK option 26

Index 257
members of partitioned data sets (see libraries)
MERGE statement restriction 179
dnames 180
message control program (MCP) 124
message format, teleprocessing 125
message processing program (MPP) 124
messages
general discussion 40
line numbers in 25
linkage editor 55
loader 70
long form option 26
numbering in preprocessor messages 43
printed format 98
severity option 24
short form option 26
sort/merge 181
statement numbers in 25
use in checking out program 157
MFT (Multiprogramming with a Fixed number of Tasks) 1,152
and cataloged procedures 152
introduction 1
MODE subparameter of DCB parameter 195
Model 195 211
Model 91 211
module map, linkage editor 56
module, load (see load modules)
linkage editor; loader 55
move mode input/output 80
MPP (message processing program) 124
MSGCLASS parameter 34
MSGLEVEL parameter 34
multiple compilation 19,47,41,62
multiple operations on punched cards 87
multiple-exception imprecise interrupt (definition) 211
multitasking
library (SYS1. PLITASK) 53,150
options in CALL PLIDUMP 162
with shared library 213
MVT (Multiprogramming with a Variable number of Tasks) 1,152
and cataloged procedures 152
introduction 1
NAME option 27,41
compatibility restriction 202
NAME statement (linkage editor) 58,27,41
names, qualified 73
NCAL option
linkage editor 54
loader 69
NCR subparameter of DCB parameter 195
NCF code for paper tape, specifying 194
NE (not editable) attribute 47
NEST option 27
nesting level in listing 36
no-operation instructions 211
NOCALL option (loader) 69
NTM subparameter of DCB parameter 196
null statement, use for controlling interrupt timing 211
NUMBER option 27
object module
combining 41
definition 2
format 19
librarians (see libraries)
listing 40
on punched cards
and cataloged procedures 153
identification 24
output 24,27
storage requirement listing 29
structure 40
OFFSET option 27,42
OFFSET option 28,38
offset variables as source of error 159
offsets, table of 28,38
on-codes 161
on-units 161
condition built-in function values 162
use in checking out program 159
ONCODE built-in function 161
ONCOUNT built-in function 211
OPEN macro instruction 81
OPEN statement 81
operating errors 159
operating system
compiler interface 15
errors 160
introduction 1
release identification 35
operating system facilities 202
OPTCD subparameter of DCB parameter 196
OPTIMIZE option 28
option list
compiler 22
dynamic invocation 45
linkage editor 53
loader 68
optional facilities
compiler 22
linkage editor 53
loader 68
OPTIONS option 28
organization of data set, specifying 195
OS facilities 202
output
(see also: data sets; input/output)
compiler 19
linkage editor 52
loader 67
overflow area
introduction 108
main discussion 112
separate DD statement for 110,114
specifying in JCL 194,196
OVERLAY statement (linkage editor) 61
overlaying
checkout of 158
library subroutines 62
linkage editor 55
main discussion 59
mapping 57
OVLY attribute (linkage editor) 61
prime data area 108,114

Index 259
read column eliminate (continued)
format descriptor card 86
RECFM subparameter of DCB parameter 196
record blocking (see: block size; blocking)
record format
auxiliary storage 80
CONSECUTIVE data sets
record I/O 101,103
stream I/O 91,95
essential parameters 80
INDEXED data sets 111
introduction 9
main discussion 75
operating system data management 80
PRINT files 95
REGIONAL data sets 118
specifying in JCL 196
record length
CONSECUTIVE data sets
record I/O 101
stream I/O 91
INDEXED data sets 110
introduction 9
PRINT files 95
REGIONAL data sets 118
specifying 195,74
record size, maximum, compiler input 19
record type specification 9
record-oriented input/output
access methods 81
defining data sets 101
record-oriented transmission 203
records 9,74
deleted (dummy)
INDEXED data sets 114,196
spanned 9,75,196
RECsize option of ENVIRONMENT attribute 74
region
definition 1
size 28,152
REGIONAL data sets 116
access 120
capacity record 118
creation 118
dummy records 118,120
register contents in trace 163
register 12, use of 169
release number in listing 35
RELEASE statement 62
relocation dictionary (RLD) 49
reply maximum length 200
REPORT option 32
REREAD option 79
RES option (loader) 69
resident control phase 15
resident library (see library subroutines)
restart 189
RESTART parameter 190
return codes
checkpoint/restart 189
compiler 41
PL/I program 163
PLIEXEC restriction 203
return code 0004 from linkage editor 56
sort/merge 177,178
returned values 201
RKP subparameter of DCB parameter 112,197
RLD (relocation dictionary) 49
root segment 59
save areas 165
scheduling time 48
scheduling, chained 80,196
segment, root 59
sequence numbering
compiler options 28
for preprocessor 43,45
SEQUENCE option 28
sequence set (VSAM) 131
sequential data sets 77
SER subparameter 10
serial number, volume (see volume serial number)
severity of messages
compiler 24
linkage editor 56
shared library cataloged procedures 213
SIGNAL statement 161
SIZE option
compiler 28,41
linkage editor 54
loader 70
SKIP option/format item 95
SMESSAGE option 26
SNAP option 160
sort/merge 177
arguments 179
data sets 178
entry names 177
examples 181
invoking 179
message listing options 181
multiple invocation 180
sorting techniques 181
storage requirements 177
user exits 177
SORTCKPT 179,180
SORTIN 178,180
SORTLIB 179
SORTOUT 179,180
SORTWK 179,180
SOURCE option 29
source program
character set specification 22
data code specification 22
data set 19
listing 25,29
compiler option for 29
nesting level 27
record numbering 25,27,28
statement numbering 29,35
source statement library 44
SPACE parameter
accessing data sets 10
for direct-access devices 89
for library 142
for linkage editor output 52
for standard data sets 19
introduction 9
spanned records 9,75,196
SPLE option 32
spill file 20
STACK subparameter of DCB parameter 197
index 261

stacker selection 105,85
   specifying in JCL 197
STAE option 32
standard files 98
statement numbers
   compiler option 29
   in messages 25
   method of numbering 35
   trace of 160
statement, maximum number of 204
restrictions 204
static internal control section
   description 48
   length 39
   listing 40
static storage map 26
STEPLIB DD statement 142
STMT option 29
storage
   addressing 49
   allocation 17
   auxiliary, economy
      blocking PRINT files 96
      in INDEXED data sets 111
      in REGIONAL data sets 118
      suppressing automatic library
         call 54
      using loader 48
      using track overflow 196
buffers 80
dumps 161
for Assembler language linkage 165,169
for checkpoint/restart data set 189
for compilation 28
for execution 31
for library data sets 142
for linkage editor 52
for loader 64,70
for sort/merge 177
for standard data sets 18
insufficient available 28
linkage editor 54
optimization 28
overlaying (see overlaying)
   requirements in general 28
   virtual 1
STORAGE option 29
stream-oriented input/output
   access method 81
   defining data sets 91
STRINGRANGE condition 160
structures
   length table 37
   mapping 17
stub, link-edit 63
subroutines, library (see library
   subroutines)
SUBSCRIPTRANGE condition 159
SUBSTR pseudovariable as source of
   error 160
supervisor (see operating system)
symbolic parameter in cataloged
   procedures 150
syntax analysis stage 17
syntax checking option
   compiler option 29
   suppression of 158
SYNTAX option 29
SYSCIN 19
SYSIN 19,98
SYSLIB
   linkage editor 52
   loader 67
   multitasking programs 150
   preprocessing 20
SYSLIN 42
   compiler output 19
   linkage editor input 51
   loader input 66
SYSLMOD 51
SYSLOUT 67,70
SYSPARM parameter 11
   INDEXED data set restriction 110
SYSPRINT
   associated with terminal 30
   compiler data set 20
   linkage editor data set 53
   loader data set 67,70
   standard PL/I file 98
SYSPUNCH 19
system failure 160
system output device (see SYSOUT parameter)
   system procedure library
      (SYS1.PROCLIB) 149
   system program library (SYS1.LINKLIB) 142
   SYSTDI
      compiler data set 20
   SYS1.LINKLIB (system program library) 142
   SYS1.PLIBASE 63,79
   SYS1.PLIBASE (base library) 53,151
   SYS1.PIICMIX 63
   SYS1.PIITASK (multitasking library) 53,150
   SYS1.PROCLIB (system procedure library) 141,149
tab control table 98
tab position specification and defaults 98
tape, magnetic (see magnetic tapes)
TCA (task communications area) 163
TCAM (Telecommunications Access Method) 81,124
Telecommunications Access Method
   (TCAM) 81,124
teleprocessing 124
teleprinter code for paper tape 194
temporary workspace
   essential parameters 52
   for compiler 20
   for linkage editor 52
TERMINAL option 30
terminal processing (see TSO)
termination
   in Assembler-PL/I linkage 169,173
   of compilation, dump option 24
   of execution, abnormal 159
   of execution, by request 162
   testing (see checkout, program)
text (TXT), description of 48
text, source (definition) 15
Time Sharing Option (see TSO)
time taken for compilation 35
timer feature 35
trace information 160,161
   compiler option 24
trace information (continued)
execution-time option 34
track (definition) 89
track index 107
track overflow, specifying 196
trailer label 77
transfer vector 213
transient control phase 15
TRANSIENT files 77
transient library (see library subroutines)
translation stages 17
TRANSIT condition, suppressing 196
troubleshooting (see checkout, program; trouble determination)
TRTCH subparameter of DCB parameter 88,197
TSO (Time Sharing Option)
checkpoint/restart restriction 189
conversation checkout 157
INTRODUCTION 1
line numbers 27
storage requirements 29
terminal output option 30
U-format records 76
unblocked records 75
undefined-length records 76
UNDEFINEDFILE condition 81,96
UNIT parameter
accessing a data set 10
creating a data set 8,78
unlabeled data sets 77
unlabeled magnetic tapes 103
unnamed data sets 74,10
updat(ing) data 10
user exit points 177
user information (see argument passed to main procedure)
V-format records 75
validity check, write, specifying 196
variable-length records 75,183
variables storage map 40
varying-length strings, sorting 178
VB-format records 75
VBS-format records 75
version number of compiler 35
virtual storage 1
volume
definition of term 9
labeling 77
VOLUME parameter
(see also volume serial number)
accessing data sets (general) 10
creating data sets (general) 10
volume serial number
accessing REGIONAL data sets 120
creating CONSECUTIVE data sets
record I/O 101,103
stream I/O 91
creating INDEXED data sets 110
creating REGIONAL data sets 118
in volume label 77
volume table of contents (VTOC) 77
VB-format records 75
VSAM (Virtual Storage Access Method) 131
VSAM data sets (continued)
catalogs 131
compatibility interface 137
control area 131
control interval 131
device restriction 131
index data set 131
index set 131
logical record 131
passwords 138
sequence set 131
sharing 138
VS1 (Virtual Storage)
INTRODUCTION 1
VS2 (Virtual Storage)
INTRODUCTION 1
VTOC (volume table of contents) 77
WAIT statement 205
weak external reference 38
workfiles, temporary 150
XCAL option (linkage editor) 55
XCTL macro instruction 45
XREF option
compiler 30
linkage editor 55
1403 Printer control characters (see printers)
2400-series tape drives, conversion
feature 197
2540 Card Read Punch control characters 95
3505 Card Reader 84
3525 Card Punch 84
48-character set 17,22
60-character set 17,22
7-track magnetic tape (see magnetic tapes)
9-track magnetic tape (see magnetic tapes)
OS
PL/I Optimizing Compiler:
Programmer's Guide
Order No. SC33-0006-3

Your views about this publication may help improve its usefulness; this form will be sent to the author's department for appropriate action. Using this form to request system assistance or additional publications will delay response, however. For more direct handling of such requests, please contact your IBM representative or the IBM Branch Office serving your locality.

Possible topics for comment are:
Clarity  Accuracy  Completeness  Organization  Index  Figures  Examples  Legibility

What is your occupation? _____________________________

Number of latest Technical Newsletter (if any) concerning this publication: __________

Please indicate in the space below if you wish a reply.

Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments.)
Your comments, please...

This manual is part of a library that serves as a reference source for systems analysts, programmers, and operators of IBM systems. Your comments on the other side of this form will be carefully reviewed by the persons responsible for writing and publishing this material. All comments and suggestions become the property of IBM.

Business Reply Mail
No postage stamp necessary if mailed in the U.S.A.

Postage will be paid by:
International Business Machines Corporation
Department 813 HP
1133 Westchester Avenue
White Plains, New York 10604