SC33-0019-1
File No. S360/S370-29

Program Product

DOS
PL/I Optimizing Compiler:
Execution Logic

IBM

SC33-0019-1
File No. S360/S370-29

Program Product

DOS
PL/I Optimizing Compiler: Execution Logic

PL/I Optimizing Compiler 5736-PL1
PL/I Resident Library 5736-LM4
PL/I Transient Library 5736-LM5
(These products are also distributed as composite package 5736-PL3)

IBM

Second Edition (September, 1973)

This is a major revision of SC33-0019-0 and associated technical newsletters.

Information has been included on the new features that are available with release 4 of the DOS PL/I Optimizing Compiler as follows:

COUNT option - Chapter 7
New options for PLIDUMP - Chapter 12

A number of minor changes and corrections have also been made throughout the book. A new topic heading "How Addressed" has been added to the control block descriptions in appendix A. Technical changes are marked with a vertical line to the left of the change.

This edition applies to Version 1, Release 4, Modification 0 of the DOS PL/I Optimizing Compiler and to all subsequent releases until otherwise indicated in new editions or Technical Newsletters. Changes are continually 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 SRL Newsletter, Order No. GA22-6822, for the editions that are applicable and current.

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 readers' 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.

Preface

The main purpose of this publication is to explain, in general terms, the way in which programs compiled by the DOS PL/I Optimizing Compiler (Program Number 5736-PL1) are executed. It describes the organization of object programs produced by the compiler, the contents of the executable program phase, and the main storage situation throughout execution. The type of information provided is intended primarily for those persons involved in program maintenance of the compiler and its related library program products, but it should also provide valuable information for applications programmers, since a knowledge of the way in which source program statements are implemented at execution time can only lead to the writing of more efficient programs. To this end, the book contains a chapter on how to obtain and read a PL/I dump.

Although different source programs produce different executable programs, the structure of every executable program phase produced by the compiler is basically the same. This structure is explained in chapter 1. Chapters 2,3,4, and 5 describe the various elements that make up the executable program phase. Chapters 6 and 7 explain the housekeeping and error-handling schemes. Chapters 8, 9, 10, and 11 describe the implementation of various language features, the majority of which are handled by a combination of compiled code, PL/I library routines, and DOS system routines. Chapter 12 is the guide to obtaining and using dumps. The final chapter, chapter 13, deals with interlanguage communication. In addition, there are two appendixes: appendix A provides a diagrammatic summary of the principal contents of main storage during program execution; appendix B contains details of all control blocks that can exist during execution.

The reader of this publication is assumed to have a sound knowledge of PL/I, and a working knowledge of the IBM Disk Operating System and its assembler language. It is recommended, therefore, that the reader should be familiar with the content of the following publications:

RECOMMENDED PUBLICATIONS

DOS PL/I Optimizing Compiler: Programmer's Guide, Order No. SC33-0008

DOS PL/I Optimizing Compiler: Language Reference Manual, Order No. GC33-0005

System/370 Principles of Operation, Order No. GA22-7000

Introduction to System Control Programs, Order No. GY24-5017

REFERENCE PUBLICATIONS

This book makes reference to the following publications for related information that is beyond its scope:

IBM System/360 Reference Data Card, Order No. GX20-1703

IBM System/370 Reference Summary, Order No. GX20-1850

IBM Disk Operating System:

DOS Supervisor and Input/Output Macros, Order No. GC24-5037

DOS PL/I Optimizing Compiler: Program Logic, Order No. LY33-6010

DOS PL/I Resident Library: Program Logic, Order No. LY33-6011

DOS PL/I Transient Library: Program Logic, Order No. LY33-6012

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.

L - Licensed materials, property of IBM: available only to licensees of the related program products under the terms of the license agreement.

Contents

Figures

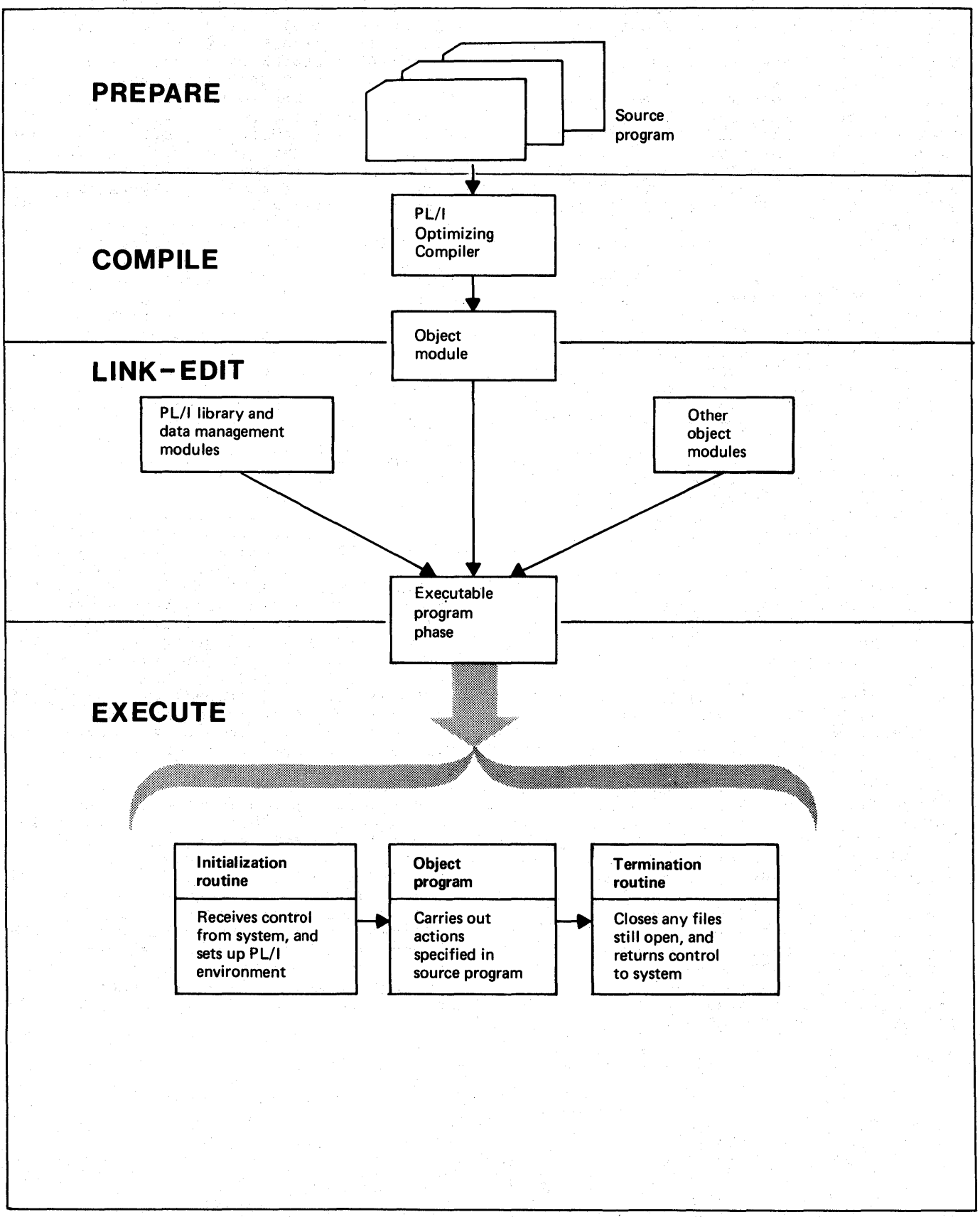


Figure 1.1. The process of running a PL/I program

Chapter 1: Introduction

Processing a PL/I Program

Figure 1.1 shows the processes through which a PL/I program passes from its inception to its use. There are four stages:

1. Writing the program and preparing it for the computer.
2. Compilation: translating the program into machine instructions (i.e., creating an object module).
3. Link-editing: producing an executable program phase from the object module. This includes linking the compiled code with PL/I library modules, data management routines, and possibly with other compiled programs. It also includes resolving addresses within the code.
4. Execution: running the executable program phase.

The process is not necessarily continuous. The program may, for example, be kept in either a compiled or link-edited form before it is executed, and it will normally be executed a number of times once compiled.

COMPILATION

Compilation is the process of translating a PL/I program into machine instructions. This is done by associating PL/I variables with addresses in storage and translating executable PL/I statements into a series of machine instructions. For example, the PL/I statements:

```
DCL I,J,K;
I=J+K;
```

would typically result in the generation of machine instructions corresponding to the assembler language instructions shown below:

```
LH    7,88(0,13)   Load J into register 7
AH    7,90(0,13)   Add K to J
STH   7,96(0,13)   Place result in I
```

(The variables I, J, and K are held at offsets 96,88, and 90, respectively, from the address in register 13.)

The DOS PL/I Optimizing Compiler does not translate all PL/I statements directly into the necessary machine instructions. Instead, certain statements are translated into calls to standard subroutines held in the DOS PL/I Resident Library. Some of the resident library routines may, in turn, call further library routines from either the resident or the transient PL/I library. The following PL/I statements would, for example, result in a call being made to a resident library routine.

```
DCL X,Y;
X=SIN(Y);
```

The code that would typically result from such statements is shown below:

```
LA     14,92(0,13)    Place address of Y
                      in register 14.
LA     15,96(0,13)    Place address of X
                      in register 15.
STM    14,15,80(0,3)  Place addresses in
                      argument list.
LA     1,80(0,3)      Point register 1 at
                      argument list.
L      15,88(0,3)     Load register 15
                      with the address of
                      the resident library
                      routine IBMBMGS.
                      (This is held in the
                      form of an address
                      constant generated
                      by the compiler and
                      resolved by the
                      linkage editor.)
BALR   14,15          Branch to the
                      library routine,
                      which will carry out
                      the required
                      function.
```

LINK-EDITING

Link-editing links the compiler output with external modules that have been requested by the compiled program. These will be PL/I resident library routines, data management routines, and, possibly, modules produced by further compilations. As well as linking the external modules, the linkage editor also resolves addresses.

EXECUTION

The optimizing compiler produces code that requires a special arrangement of control blocks and registers for correct execution. This arrangement of control blocks and registers is known as the PL/I environment. Execution consequently becomes a three-stage process:

1. Setting-up the environment. This is handled by the PL/I initialization routines IBMDPIR and IBMDPII.
2. Executing the program.
3. Completing jobs after execution. This consists of closing any files that are left open and either returning control to the control program, with an EOJ macro instruction, or returning to a calling module.

Factors Affecting Implementation

Three major factors influenced the design of the executable programs produced by the optimizing compiler. These factors are inherent in the language, and are:

1. The modular structure of PL/I programs

   The PL/I language allows the programmer to divide his program into a series of blocks that can be written and compiled independently of each other.

2. The dynamic allocation and freeing of storage

   Automatic, controlled, and based variables all have their storage allocated and freed dynamically. This implies a system of re-use of storage to reduce space requirements.

3. The comprehensive facilities offered by the PL/I language

   The PL/I language offers more facilities than any other high-level language. These facilities include allowing the PL/I program to control the flow of execution after any PL/I interrupt.

Key Features of the Executable Program

Taken together, the factors outlined above are responsible for the main features of the executable program produced by the compiler. These features are:

1. A communications area addressed by a dedicated register throughout the execution of the program.
2. A scheme to handle dynamic storage allocation.
3. The use of standard subroutines from the PL/I libraries, to handle such standard tasks as the housekeeping scheme and error handling.
4. The use of an initialization routine to set up the communications area and initiate the housekeeping scheme.

These features are discussed in greater detail below.

COMMUNICATIONS AREA

The facilities offered by the PL/I language, particularly the error-handling facilities, imply that certain items must be accessible at all times during execution. To simplify accessing such items, a standard communications area is set up for the duration of execution. This area is known as the task communications area (TCA), and is addressed by register 12 throughout execution.

The TCA has an appendage known as the task implementation appendage (TIA). The TCA appendage holds a number of addressing fields and is, itself, addressed from the TCA.

DYNAMIC STORAGE ALLOCATION

The allocation and freeing of automatic storage on a block-by-block basis implies an automatic facility for the re-use of such storage. This problem and the problem of inter-block communication are solved by having, for each block, a save area that contains register save information, automatic variables, and housekeeping information. This area is known as a dynamic storage area (DSA). It consists of the standard operating system save area concatenated with certain housekeeping information and with storage for automatic variables. DSAs are held contiguously in a last-in/first-out (LIFO) storage stack and are freed and allocated by the alteration of pointer values.

On entry to a block, the registers of

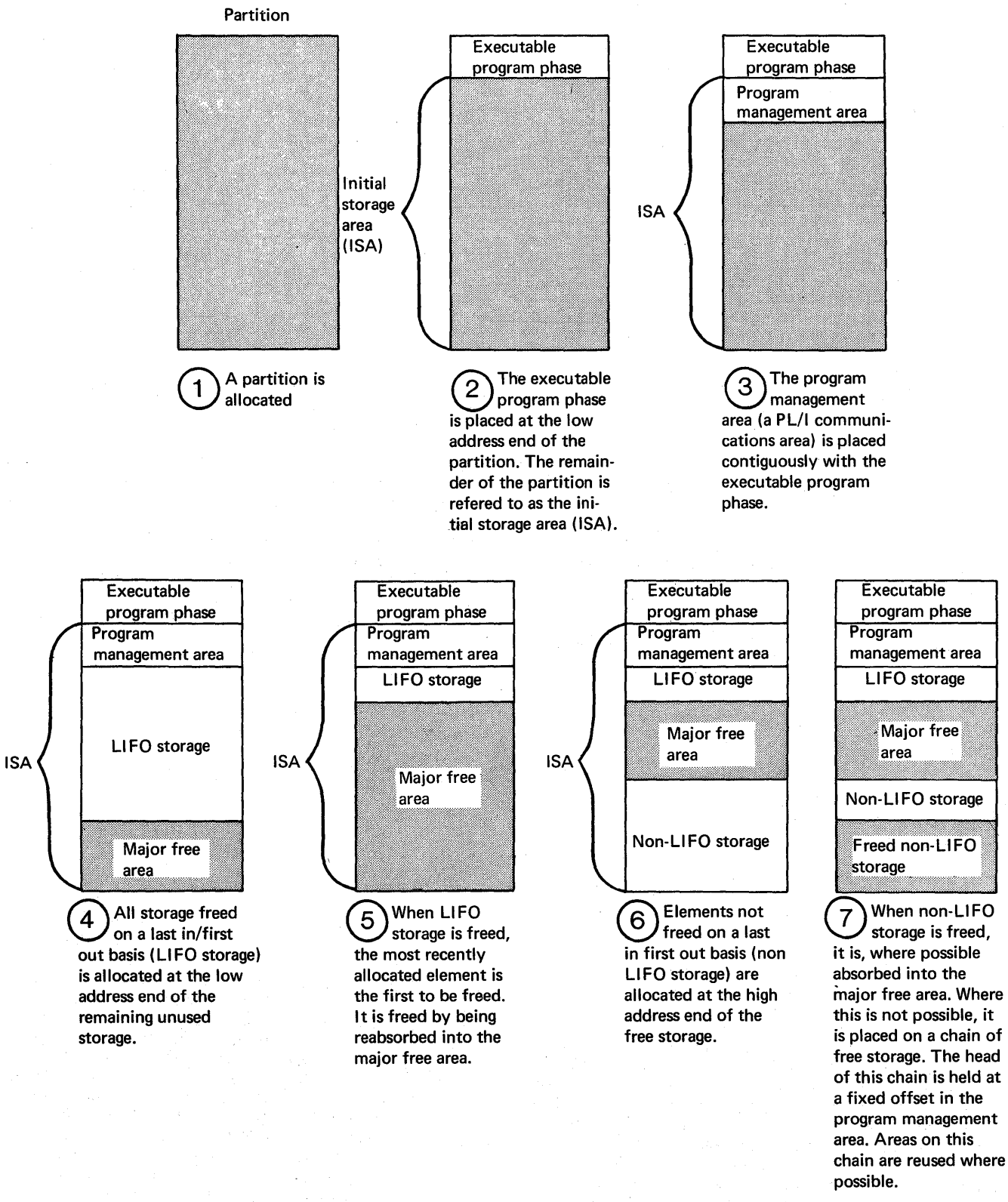


Figure 1.2. Use of dynamic storage

the preceding block are stored in the previous DSA and a new DSA is acquired. A chainback pointer to the previous DSA is placed in the new DSA. This arrangement allows access to information in previous blocks. Register 13 is pointed at the head of the DSA for the current block. The code that carries out this and any other block initialization is known as a prologue. To obviate the need for special coding in the main procedure, a dummy DSA is set up by an initialization routine, and register 13 points at this dummy DSA on entry to the main procedure.

In addition to automatic variables, certain other types of storage are allocated and freed dynamically. Such items as are not freed on a last-in/ first-out basis are kept in a second stack. If storage within this stack is freed, it is placed on a free-area chain. The principles of the dynamic storage scheme are illustrated in figure 1.2.

In certain circumstances, additional LIFO storage may be required during the execution of a block. When this is necessary storage is acquired in the same manner as for a DSA. The areas thus acquired are known as variable data areas (VDAs).

The storage scheme is handled partly by compiled code and partly by a resident-library routine. Compiled code acquires and frees space in the LIFO storage stack. LIFO storage is acquired by the prologue code of every block and freed by the epilogue code of every block.

The library routine IBMDPGR is called when non-LIFO dynamic storage has to be allocated or freed, or when there is insufficient space for an allocation of LIFO storage in the LIFO stack.

USE OF LIBRARY SUBROUTINES

The use of library subroutines simplifies compilation. However, using such routines slows execution because they cannot be tailored for the particular situation in hand, and because they incur the overhead of saving and restoring registers. Library subroutines are used for handling standard jobs such as program initialization and error handling, and for such items as require interpretive code. Interpretive code is required when a significant part of the data will not be available until execution.

Two PL/I libraries are used by the DOS PL/I Optimizing Compiler: the DOS PL/I Resident Library and the DOS PL/I Transient Library. Transient library routines have the advantage of saving space, because they require storage only when they are actually in use and can be overwritten when they are no longer required. Resident library routines, however, have the advantage of speed, because they do not have to be loaded during execution of the PL/I program. Dividing subroutines into transient and resident types enables the compiler to balance the advantages of both types and so to produce programs that combine fast execution with reduced space overheads.

INITIALIZATION ROUTINES

The job of the initialization routines is to prepare a standard environment for all procedures compiled by the DOS PL/I Optimizing Compiler. This consists of setting-up the TCA and initializing the storage scheme. Also, a STXIT macro instruction is issued so that all program checks will be intercepted by the PL/I error-handling facilities. Using standard library routines for these tasks reduces the amount of special-case coding that is needed for a main procedure. A consequence is that procedures can be compiled and tested individually and then link-edited with other procedures and run without re-compilation.

Contents of a Typical Executable Program Phase

The contents of a typical executable program phase are shown in figure 1.3. The contents are:

1. Compiled code (the executable machine instructions that have been generated).

2. Link-edited routines. These will include resident library routines and probably DOS data management routines. Certain resident library routines are included in every executable program phase. These are the initialization routine, IBMDPIR, the storage-handling routine, IBMDPGR, and the error handler, IBMDERR. Other resident routines are included as required.

As well as executable machine instructions, the program requires certain control information and addresses. Some of these are listed in figure 1.3, but full details are given in chapter 2. Other

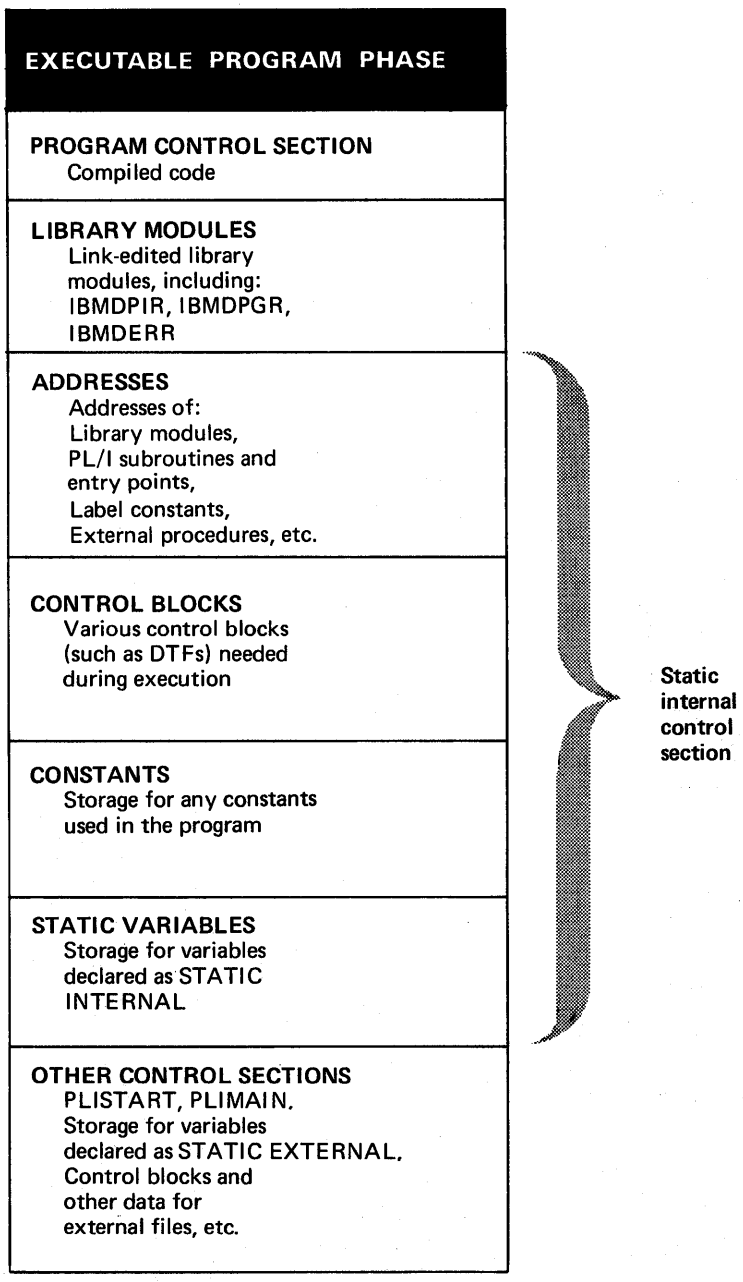


Figure 1.3. Simplified diagram of an executable program phase

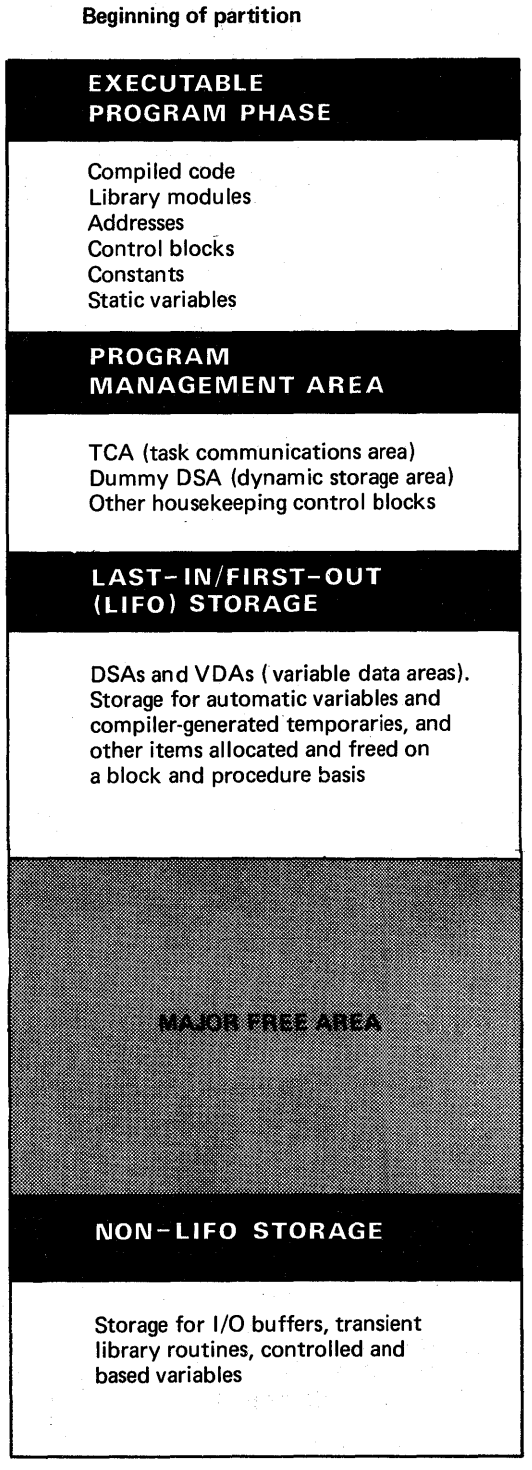


Figure 1.4. Use of storage

control sections generated are also shown in figure 1.3. They are PLISTART, which passes control to the initialization routine, and PLIMAIN, which holds the address of the start of compiled code.

The Overall Use of Storage

The overall use of storage is illustrated in figure 1.4. As can be seen, the low-address end of the partition is occupied by the executable program phase. Immediately following the executable program phase is the program management area. This contains the control blocks set up by the initialization routines, including the TCA and the dummy DSA discussed above. The remainder of the partition is used for dynamic allocations of storage. The LIFO stack starts beyond the end of the program management area and expands, as necessary, towards the end of the partition. Non-LIFO dynamic storage starts at the end of the partition and expands towards the LIFO stack.

The Process of Execution

The process of execution is illustrated in figure 1.5. The processes involved for a sample program are described below.

```
EXAMPLE:  PROC OPTIONS(MAIN);
   INPUT:  GET LIST(Y,Z);
             .
             .
             .
       (process data as required)
             .
             .
             .
           PUT LIST(X);
           IF X<500 THEN GO TO INPUT;
           END;
```

Execution would involve the steps described below.

1. The control program passes control to the control section PLISTART, which has been generated by the compiler.

2. PLISTART calls the resident library initialization routine, IBMDPIR.

3. IBMDPIR, and IBMDPII, which it calls, set up the PL/I environment. IBMDPIR then passes control to the main procedure compiled code, with register 12 pointing at the TCA and register 13 pointing at the dummy DSA. The address to which IBMDPIR passes

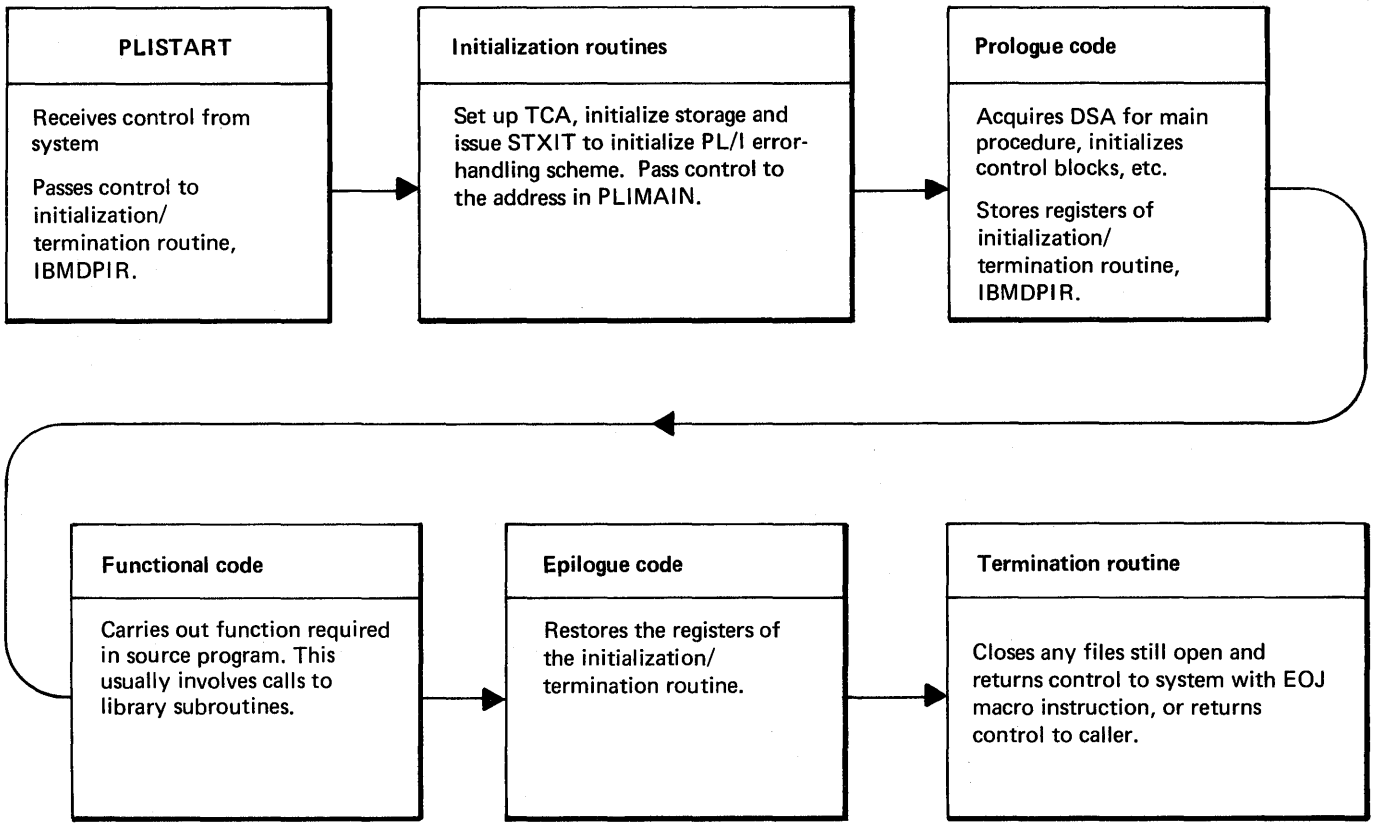


Figure 1.5. Flow of control during execution

control is held in the control section PLIMAIN.

4. Compiled code prologue stores the contents of the registers used by IBMDPIR in the dummy DSA and acquires a DSA for the main procedure.

5. Compiled code calls the library routines used for stream I/O. These in turn call transient routines to open the standard files and further transient routines to interface with, and call, the link-edited data management routines. Storage must be acquired for transient routines and I/O buffers. This involves calling the storage management routine IBMDPGR.

6. Processing is then carried out by compiled code. Further calls to the library may be involved if, for example, mathematical functions are used.

7. The stream output will involve further steps similar to those described in 5, above.

8. When the END statement is reached, the epilogue code is entered. This restores the registers of IBMDPIR and returns control to IBMDPIR.

9. IBMDPIR raises the FINISH condition, calling the resident error-handling module IBMDERR, which searches for a FINISH on-unit. Finding none, it returns to IBMDPIR; IBMDPIR calls IBMDOCL to close the standard files SYSIN and SYSPRINT, which were opened to permit execution of the stream I/O statements. An EOJ macro instruction is then issued to terminate the program.

This program illustrates the main points mentioned earlier in the chapter. The initialization routines are used in steps 3 and 9. The storage management scheme is illustrated in the prologue and epilogue code in steps 4 and 8. The communications area (TCA) is set up by the initialization routine, and the use of standard library subroutines is shown in steps 5 and 7. The use of special error and PL/I condition handling code is shown in step 9.

Chapter 2: Compiler Output

Introduction

The compiler output is a relocatable object module consisting of a series of records in card-image format. These records contain either machine instructions, constants, or external or internal addresses to be resolved by the linkage editor. The records are known as:

TXT records containing machine instructions or constants.

RLD records containing internal addresses.

ESD records containing external addresses.

Further information about the output passed to the linkage editor is given in the publication DOS PL/I Optimizing Compiler Program Logic.

There are two main control sections (CSECTs) output by the compiler. These are:

1. The program control section, holding the executable instructions translated from the PL/I program.

2. The static internal control section holding constants, addresses, and static variables.

A number of other control sections are also generated. These either handle certain housekeeping functions, or are used for external data which may have identical control sections generated for it by other compilations.

Workspace and storage for automatic variables is acquired during execution, normally by the prologue code that is executed at the start of every block.

The output from the compiler is shown in figure 2.1 and listed below:

1. Control sections that are always generated

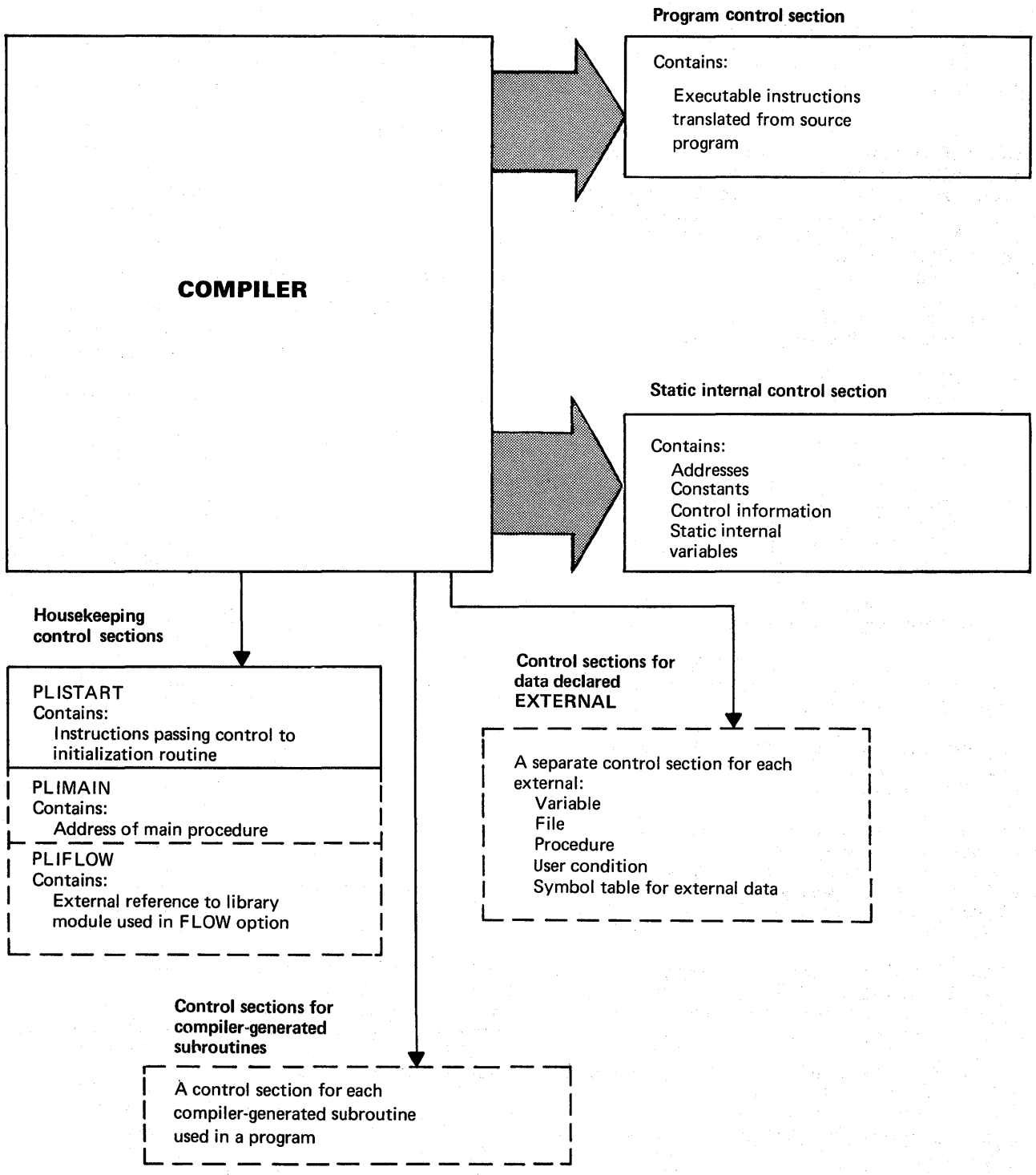
Program control section	Containing executable instructions.
Static internal control section	Containing addresses, control blocks, constants, and STATIC INTERNAL variables.
PLISTART	The entry point for the executable program phase. Passes control to initialization routine.

2. Control sections that are generated only when required

PLIMAIN	Containing the address of the entry point of the main procedure. (Generated only for procedures with OPTIONS(MAIN).)
PLIFLOW	A control section generated when the compiler FLOW option is specified. (See chapter 7.)
PLICOUNT	A control section generated when the compiler COUNT option is specified. (See chapter 7).
Static external control sections	A static external control section is generated for every external variable, file, and procedure.
Plus control sections for	Each user-defined condition, and each compiler-generated subroutine used.

The remainder of this chapter deals with these control sections in further detail. Where possible, it refers to the object program listing, because this is the form in which the output from the compiler is most readily available.

The two control sections, PLISTART and PLIMAIN, are used during program initialization. PLISTART holds the address of the library initialization routine IBMDPIR, which will be entered at the start of the program. PLIMAIN holds the address of the start of the code for the main procedure. This is the address to which the library initialization routine branches when initialization is complete; it is marked "*REAL ENTRY" in the object-program listing.



Control sections surrounded with dotted lines are generated only when required.

Figure 2.1. Output from the compiler

A PLIMAIN control section is generated for every procedure for which OPTIONS (MAIN) is specified in the procedure statement. When two such procedures are being run together, control will always pass to the first of the procedures processed by the linkage editor.

The format of PLIMAIN and PLISTART is given in appendix B.

If the compiler FLOW option is being used, a control section called PLIFLOW is also generated. This contains code that results in the link-editing of the trace module IBMDEFL and also contains the values of "n" and "m" specified in the option. The format of PLIFLOW is given in chapter 7.

If the compiler COUNT option is in effect, a control section called PLICOUNT is generated. This contains code to link-edit IBMDEFL.

The Organization of this Chapter

The remainder of this chapter describes the contents of the static internal control section and the program control section. First, the conventions used in the object program listing and the static storage map are described. Descriptions of the two control sections follow. The description of the program control section covers the conventions used in the object program code such as register usage, method of handling flow of control, and addressing information. The chapter is completed by a short discussion of the effects of optimization.

Listing Conventions

Figure 2.2 shows the major program listing information that can be produced by the compiler. It also shows the relevant compiler options and summarizes the information that will be produced if these options are specified. Some or all of these options may be deleted at system generation time. To obtain deleted options, the correct password (specified at system generation time) must be specified in the CONTROL option.

This chapter describes the contents of the static-storage map and the object-program listing. Information on the other items generated is given in the publication DOS PL/I Optimizing Compiler Programmer's Guide.

STATIC-STORAGE MAP

The static-storage map is a formatted listing of the contents of the static internal and static external control sections. The static control sections contain items grouped in the following order:

1. Address constants for entry points to procedures, and for branch instructions.
2. Address constants for resident library subroutines.
3. Address constants for addressing static storage beyond 4K.
4. The constants pool, which contains source program constants, data element descriptors, locator/descriptors, symbol tables, file control blocks, and other control blocks.
5. Static variables.

The constants pool and the static-variable sections of static storage begin on doubleword boundaries.

The static control section is listed, each line comprising the following elements:

1. Six-digit hexadecimal offset.
2. Hexadecimal text, in 8-byte sections where possible.
3. Comment, indicating the type of item to which the text refers; a comment appears against only the first line of the text for an item. A typical example is shown in figure 2.3.

The following comments are used (xxx indicates the presence of an identifier):

A.. - Address constant.

COMPILER LABEL CL.nn - Compiler-generated label.

CONDITION CSECT - Control section for programmer-named condition

CONSTANT

CSECT FOR EXTERNAL VARIABLE - Control section for external variable.

D.. - Descriptor.

DED.. - Data element descriptor.

DTF (CONSTANT PART) - Constant part of

Name	Contents	Compiler Option
Source program	Source program statements	SOURCE
Aggregate table	Names and storage requirements of structures and arrays	AGGREGATE
Storage requirements	Names and storage requirements of all procedures	STORAGE
ESD references	Name, type, and identifier of all external symbols generated by the compiler*	ESD
Statistics	Number of source records, program text statements, and object code bytes	ESD
Static storage	Contents of static internal and static external control sections in hexadecimal notation with comments	MAP
Variables offset map	The offset of static and automatic variables from this defining base	MAP
Table offset and statement number	Offsets, within code, of the start of all statements	OFFSET
Object program	The contents of the program control section in hexadecimal and translated into a pseudo-assembler-language format	LIST

* External references within library modules are not included.

Figure 2.2. Contents of listing and associated compiler options

define-the-file (data management) control block.

DTF (VARIABLE PART) - Variable part of define-the-file control block.

ENVB - Environment control block.

FCB - File control block.

FED.. - Format element descriptor.

KD.. - Key descriptor.

ONCB - ON control block.

PICTURED DED.. - Pictured DED.

RD.. - Record descriptor.

SYMTAB - Symbol table.

USER LABEL xxx - Source program label xxx.

xxx - Name of static variable. If the variable is not initialized, no text appears against the comment; there is also no static offset if the variable is an array. (The static offset can be calculated from the array descriptor if required.)

OBJECT-PROGRAM LISTING

By including the option LIST in the PROCESS statement, the programmer can obtain a listing of the compiled code, known as the object-program listing. It consists of the machine instructions plus an interpretation of these instructions in a form that resembles assembler language, and a number of comments such as the statement number. The format of this listing is shown in figure 2.4. As can be seen, blocks of code are headed by the number of the statement in the PL/I program to which they are equivalent. When optimization has resulted in code being moved out of a statement, this is indicated. Only executable statements appear in the listing. DECLARE statements are not included, because they have no direct machine-code equivalent. To simplify understanding of the listing, the names of PL/I variables are inserted, rather than the addresses that appear in the machine code. Special mnemonics are used when referring to control blocks and

```
                        STATIC INTERNAL STORAGE MAP

CC0000  00000098          PROGRAM ACCON
000004  00000008          PROGRAM ACCON
000008  000000[5]A         PROGRAM ACCON
00000C  00000064          PROGRAM ACCON
000010  00000064          PROGRAM ACCON
000014  00000000          A..IELGGIA
000018  00000000          A..IBMBCACA
00001C  00000000          A..IBMBCEDB
000020  00000000          A..IBMBCHFC
000024  00C00000          A..IBMBCTHC
000028  00000000          A..IBMBCVDY
00002C  00C00000          A..IBMBDCLA
000030  00C00000          A..IBMBDCLC
000034  00000000          A..IBMBSEIA
000038  00000000          A..IBMBSICA
00003C  00C00000          A..IBMBSXCA
000040  00000000          A..IBMBSMWA
000044  00000000          A..IBMBSEEA
000048  00C00000          A..IBMBSEHA
00004C  0C000000          A..IBMBSILA
CC0050  C8040680          DED..X
000054  500000030080      FED
00005A  60C00008          FED
00005E  58010000          FED
000062  000A              CONSTANT
000064  0001              CONSTANT
000066  0004              CONSTANT
000068  91EG91E0          CONSTANT
00006C  00C00000          CONSTANT
0C0070  46008000          CONSTANT
000074  00000000          A..FCB
0C0078  C0C00000          A..FCB
0C007C  00000000          A..FCB
0CC08C  80000000          A..TEMP
000084  00000000          A..FCB
000088  80000000          A..TEMP
0CC08C
0C009C  000000FC00000064  COMPILER LABEL CL.11


                        STATIC EXTERNAL CSECTS

CC000C  000000C000000000  FCB
        000C0C0000C0C000
        0000007000000078
        000000A8000C0000

        4040000001100000
        808C0C2002C00000
        0050E200000C0050
        00C0019800000198
        0000000000400[0]E8
        C0C000C000000000
        00CC000000000050
        0000000000000000
        00C0006000000000
        00C0000600C000CC
        0005E2E8E2C9D51F
000C78  00C0000002000[9]C   ENVB
        010000A[0]02CC009C
        0200009C0200009C
        02C0009C02C0009C
        0200009C
000C9C  00C00000            ENVB CONSTANT
000CA0  00000050            ENVB CONSTANT
000CA8  00C08000[0]8000001  DTF (CONSTANT PART)
0C0CBC  C0000100            DTF (VARIABLE PART)
000CB4  00000000[0]00000000 DTF (CONSTANT PART)
        33C2E2E8E2C5D540
        400C0000[0]00C000C0
        00C0C800002020F3
000[0]D4  800C0[1]98        DTF (VARIABLE PART)
000CD8  800CC00000CC0000    DTF (CONSTANT PART)
        C0C000000000CFF00
        C0CCCC0000C000CC
        130100200C000CC
        0000005047000[0]C0
000100  07C00E2             DTF (VARIABLE PART)
000104  40000006            DTF (CONSTANT PART)
000108  31[0]000E4          DTF (VARIABLE PART)
00010C  40C00005            DTF (CONSTANT PART)
000110  080C0108            DTF (VARIABLE PART)
000114  20000001            DTF (CONSTANT PART)
000118  06CC0198            DTF (VARIABLE PART)
00011C  200000[5]005CCD858  DTF (CONSTANT PART)
        [6]0C0007931CCDDB4
        400000050800DDF8
        20000[0]011ECCDECE
        3000008100000000
        000C000000C00000
        C0C0000000000C00
        000C000000000000
        000C0C0000C00000
        C0CC000000000000
        00CC000000CC000C
        0000000000000000
        000C000C0000000C
        C0C000000C000000
        000000047FF001C
```



```
        47FF001C

000CCC  C0C0CCCC000C0000  FCB
        C000[0]C000CC00000
        000000706000007C
        C0000CA8CC0C0000
        4040000041201000
        80800C2002000000
        0C79E2C00000CC079
        000001980000C0198
        000000C00C000000
        C0C00CC0000C0000
        00C000000000C0078
        C0C0CCC00C0C0000
        00C000C0000C0000
        00C0000000C00000
        0008E2E8E2D7D9C9
        D5E30000
00007C  C000C0G0020000A0  ENV8
        010000A4G20C00A0
        020000A00020C0A0
        020000A0020000A0
        020000A0
0C00A0  00000000          ENVB CONSTANT
0C00A4  00000079          ENVB CONSTANT
CC00A8  000080C0G8000003  DTF (CONSTANT PART)
000080  00000100          DTF (VARIABLE PART)
000084  00C0000000000000  DTF (CONSTANT PART)
        3300E2E8E2D7D9C9
        D5C00CC000000000
        000008000002020F3
C000D4  240CC198          DTF (VARIABLE PART)
0C00D8  80000CC000C00000  DTF (CONSTANT PART)
        00C000C00000CFF00
        C0C000C00C00C000
        1300002000000000
        000000794700[0]000
0C010C  C7000CE2          DTF (VARIABLE PART)
C00104  400000[0]6        DTF (CONSTANT PART)
000108  310000E4          DTF (VARIABLE PART)
C0010C  40C00005          DTF (CONSTANT PART)
00011C  08000108          DTF (VARIABLE PART)
000114  20C00001          DTF (CONSTANT PART)
CC0118  1D0000F4          DTF (VARIABLE PART)
C0011C  A0000008          DTF (CONSTANT PART)
C00120  C5C00198          DTF (VARIABLE PART)
000124  60000079          DTF (CONSTANT PART)
000128  310000E4          DTF (VARIABLE PART)
00012C  40000005          DTF (CONSTANT PART)
CC0130  08000128          DTF (VARIABLE PART)
000134  20C00001          DTF (CONSTANT PART)

000138  1E0C0118          DTF (VARIABLE PART)
00013C  300C0081[0]0CC0CCC  DTF (CONSTANT PART)
        000C000000000000
        C0CC00C000CC00C0
        C00CC0000CCC00C0
        C0CCC0000000000C
        0CCC000000C00000
        C0CC0C0000C00CC0
        000000C0C00C0CC0
        00C0C0C0G00CC0C0
        C0CC0C0000C000C0
        0CCC000047FF001C
        47FF001C
```

SOURCE LISTING

```
STMT LEV NT

   1      0    EXAMPLE: PROC OTIONS (MAIN) REORDER;
   2    1 0              DCL X(10),Y,Z INITIAL (0);

   3    1 0              GET EDIT(X,Y)(F(3),X(11));

   4    1 0                DO I = 1 TO Y;
   5    1 1                  Z=Z+X(I);
   6    1 1                END;

   7    1 0              PUT EDIT(Z)(A);

   8    1 0            END;
```

Figure 2.3. Example of static storage map

```
  OBJECT LISTING

* COMPILER GENERATED SUBROUTINE IELCGIA
000000  50 E0 1 004              ST    14,4(0,1)
000004  58 F0 1 014              L     15,20(0,1)
000008  D2 03 1 01C D 04C        MVC   28(4,1),76(13)
00000E  91 10 1 011              TM    17(1),X'10'
000012  47 10 7 01A              BO    **8
000016  96 04 C 002              OI    2(12),X'04'
00001A  D5 01 F 050 E 002        CLC   80(2,15),2(14)
000020  47 40 7 04E              BL    **46
000024  91 4C F 02C              TM    44(15),X'40'
000028  47 80 7 030              BZ    **8
00002C  96 80 1 010              OI    16(1),X'80'
000030  48 7C F 050              LH    7,80(0,15)
000034  48 70 E 002              SH    7,2(0,14)
000038  40 70 F 050              STH   7,80(0,15)
00003C  58 70 F 04C              L     7,76(0,15)
000040  50 70 1 000              ST    7,0(0,1)
000044  4A 70 E 002              AH    7,2(0,14)
000048  50 70 F 04C              ST    7,76(0,15)
00004C  07 F6                    BR    6
00004E  58 FC 7 064              L     15,100(0,7)
000052  95 60 E 000              CLI   0(14),X'60'
000056  47 70 7 05E              BNE   *+8
00005A  58 F0 7 068              L     15,104(0,7)
00005E  05 EF                    BALR  14,15
000060  07 F6                    BR    6
000062  07 00                    NOPR  0
000064                           DC    AL4(0)
000068                           DC    AL4(0)

* END OF COMPILER GENERATED SUBROUTINE


* STATEMENT NUMBER  1
000000                           DC    C'EXAMPLE'
000007                           DC    AL1(7)

* PROCEDURE                             EXAMPLE

* REAL ENTRY
000008  90 EC D 00C              STM   14,12,12(13)
00000C  47 F0 F 010              B     **12
000010  00000120                 DC    F'288'
000014  00000000                 DC    A(STATIC CSECT)
000018  58 3C F 00C              L     3,12(0,15)
00001C  58 10 D 04C              L     1,76(0,13)
000020  58 00 F 008              L     0,8(0,15)
000024  1E C1                    ALR   0,1
000026  55 0C C 00C              CL    0,12(0,12)
00002A  47 D0 F 02C              BNH   **10
00002E  58 FC C 074              L     15,116(0,12)
000032  05 EF                    BALR  14,15
000034  58 E0 D 048              L     14,72(0,13)
000038  18 F0                    LR    15,0
00003A  90 EC 1 048              STM   14,0,72(1)
00003E  50 D0 1 004              ST    13,4(0,1)
000042  41 C1 0 000              LA    13,0(1,0)
000046  50 5C D 058              ST    5,88(0,13)
00004A  92 80 D 000              MVI   0(13),X'80'
00004E  92 20 D 001              MVI   1(13),X'20'
000052  D2 03 D 054 3 068        MVC   84(4,13),104(3)
000058  05 20                    BALR  2,0

* PROLOGUE BASE

* INITIALISATION CODE FOR Z
00005A  78 40 3 06C              LE    4,108(0,3)
00005E  70 40 D 0AC              STE   4,Z
* END OF INITIALISATION CODE FOR Z

000062  05 20                    BALR  2,0

* PROCEDURE BASE

* STATEMENT NUMBER  3
000064  41 40 D 0F8              LA    4,248(0,13)
000068  50 40 3 080              ST    4,128(0,3)
00006C  96 80 3 080              OI    128(3),X'80'
000070  92 24 D 109              MVI   265(13),X'24'
000074  41 E0 3 090              LA    14,144(0,3)
000078  50 E0 D 110              ST    14,272(0,13)
00007C  41 10 3 07C              LA    1,124(0,3)
000080  58 F0 3 04C              L     15,A..IBMBSILA
000084  05 EF                    BALR  14,15
000086  41 A0 2 072              LA    10,CL.10
00008A  48 E0 3 064              LH    14,100(0,3)
00008E  50 E0 D 0E0              ST    14,224(0,13)
000092                     CL.5  EQU   *
000092  58 40 D 0E0              L     4,224(0,13)
000096  88 40 0 002              SLA   4,2
00009A  41 E4 D 0B4              LA    14,VD..X(4)
00009E  41 F0 3 050              LA    15,DED..VD..X
0000A2  41 10 D 0F8              LA    1,248(0,13)
0000A6  50 10 D 0E4              ST    1,228(0,13)
0000AA  90 EF 1 008              STM   14,15,8(1)
0000AE  05 AA                    BALR  10,10
0000B0  58 E0 D 0E0              L     14,224(0,13)
0000B4  4A E0 3 064              AH    14,100(0,3)
0000B8  50 E0 D 0E0              ST    14,224(0,13)
0000BC  49 E0 3 062              CH    14,98(0,3)
```



```
0000C0  47 C0 2 02E              BNH   CL.5
0000C4  41 E0 D 0A8              LA    14,Y
0000C8  41 F0 3 050              LA    15,DED..Y
0000CC  90 EF 1 008              STM   14,15,8(1)
0000D0  05 AA                    BALR  10,10
0000D2  47 F0 2 098              B     CL.11
0000D6                     CL.10 EQU   *
0000D6  41 E0 3 054              LA    14,84(0,3)
0000DA  58 10 D 0E4              L     1,228(0,13)
0000DE  50 E0 1 004              ST    14,4(0,1)
0000E2  58 FC 3 044              L     15,A..IBMBSEEA
0000E6  05 EF                    BALR  14,15
0000E8  05 AA                    BALR  10,10
0000EA  41 E0 3 05A              LA    14,90(0,3)
0000EE  58 10 D 0E4              L     1,228(0,13)
0000F2  58 70 3 014              L     7,A..IELCGIA
0000F6  05 67                    BALR  6,7
0000F8  47 F0 2 072              B     CL.10
0000FC                     CL.11 EQU   *


* STATEMENT NUMBER  4
0000FC  78 00 D 0A8              LE    0,Y
000100  70 00 D 0E8              STE   0,232(0,13)
000104  48 70 3 064              LH    7,100(0,3)
000108  4C 70 D 0B0              MH    7,I
00010C  48 40 D 0B0              LH    4,I
000110  50 40 D 0F8              ST    4,248(0,13)
000114  48 40 3 070              LH    4,112(0,3)
000118  40 40 D 0F8              STH   4,248(0,13)
00011C  97 80 D 0FA              XI    250(13),X'80'
000120  78 2C D 0F8              LE    2,248(0,13)
000124  7B 2C 3 070              SE    2,112(0,3)
000128  70 20 D 0EC              STE   2,236(0,13)
00012C  39 20                    CER   2,0
00012E  47 20 2 116              BH    CL.3
000132                     CL.2  EQU   *


* STATEMENT NUMBER  5
000132  48 9C D 0B0              LH    9,I
000136  8B 9C 0 002              SLA   9,2
00013A  78 4C D 0AC              LE    4,Z
00013E  7C 49 D 0B4              ME    4,VD..X(9)
000142  70 40 D 0AC              STE   4,Z


* STATEMENT NUMBER  6
000146  48 7C D 0B0              LH    7,I
00014A  4A 70 3 064              AH    7,100(0,3)
00014E  40 70 D 0B0              STH   7,I

* CODE MOVED FROM STATEMENT NUMBER  4
000152  48 70 D 0B0              LH    7,I
000156  50 7C D 0F8              ST    7,248(0,13)
00015A  48 70 3 070              LH    7,112(0,3)
00015E  40 70 D 0F8              STH   7,248(0,13)
000162  97 80 D 0FA              XI    250(13),X'80'
000166  78 CC D 0F8              LE    0,248(0,13)
00016A  7B CC 3 070              SE    0,112(0,3)
00016E  70 CC D 0F0              STE   0,240(0,13)
000172  79 CC D 0E8              CE    0,232(0,13)
000176  47 CC 2 0CE              BNH   CL.2

* CONTINUATION OF STATEMENT NUMBER  6
00017A                     CL.3  EQU   *


* STATEMENT NUMBER  7
00017A  41 40 D 0F8              LA    4,248(0,13)
00017E  50 4C 3 088              ST    4,136(0,3)
000182  96 8C 3 088              OI    136(3),X'80'
000186  92 20 D 109              MVI   265(13),X'20'
00018A  41 10 3 084              LA    1,132(0,3)
00018E  58 FC 3 038              L     15,A..IBMBSIDA
000192  05 EF                    BALR  14,15
000194  41 A0 2 14E              LA    10,CL.7
000198  41 E0 D 0AC              LA    14,Z
00019C  41 F0 3 050              LA    15,DED..Z
0001A0  41 10 D 0F8              LA    1,248(0,13)
0001A4  50 1C D 0E4              ST    1,228(0,13)
0001A8  90 EF 1 000              STM   14,15,0(1)
0001AC  05 AA                    BALR  10,10
0001AE  47 F0 2 166              B     CL.8
0001B2                     CL.7  EQU   *
0001B2  41 E0 3 05E              LA    14,94(0,3)
0001B6  58 1C D 0E4              L     1,228(0,13)
0001BA  50 EC 1 00C              ST    14,12(0,1)
0001BE  58 F0 3 048              L     15,A..IBMBSEHA
0001C2  05 EF                    BALR  14,15
0001C4  05 AA                    BALR  10,10
0001C6  47 F0 2 14E              B     CL.7
0001CA                     CL.8  EQU   *


* STATEMENT NUMBER  8
0001CA  18 CD                    LR    C,13
0001CC  58 EC D 004              L     13,4(0,13)
0001D0  58 EC D 00C              L     14,12(0,13)
0001D4  98 2C D 01C              LM    2,12,28(13)
0001D8  05 1E                    BALR  1,14

* END PROCEDURE
0001DA  07 C7                    NOPR  7
```

Figure 2.4. Example of object program listing

other items.

Statements in the object program listing are ordered by block. Statements in the outermost block are given first, followed by statements in the inner blocks. Thus the order of statements will frequently differ from that of the source program.

Every object-program listing begins with the name of the procedure. The name is defined as a constant in a DC instruction. This is followed by another constant containing the length of the procedure name. Then comes the name of the procedure, as a comment, followed by code under the heading "REAL ENTRY." This is the point at which the code will, in fact, be entered. The second section of code is the prologue, which carries out various housekeeping tasks and is described more fully later in this chapter. The end of the prologue is marked by the message "PROCEDURE BASE." This is followed by a translation of the first executable statement in the PL/I source program.

The comments used in the listing are as follows:

* PROCEDURE xxx - identifies the start of the procedure labeled xxx.

* REAL ENTRY xxx - heads the initialization code for an entry point to a procedure labeled xxx.

* PROLOGUE BASE - identifies the start of the prologue code common to all entry points into that procedure.

* PROCEDURE BASE - identifies the address loaded into the base register for the procedure.

* STATEMENT LABEL xxx - identifies the position of source program statement label xxx

* PROGRAM ADDRESSABILITY. REGION BASE - identifies address to which the program base is updated if program exceeds 4096 bytes and cannot be addressed from one base.

* CONTINUATION OF PREVIOUS REGION - identifes the point at which addressing from the previous program base recommences.

* END OF COMMON CODE - identifies the end of code used in the execution of more than one statement.

* END PROCEDURE xxx - identifies the end of the procedure labeled xxx.

* BEGIN BLOCK xxx - indicates the start of the begin block with label xxx.

* END BLOCK xxx - indicates the end of the begin block with label xxx.

* BEGIN BLOCK - GENERATED NAME BLOCK.nn - indicates the start of an unnamed begin block for which the compiler has generated the name BLOCK.nn, where nn is two hexadecimal digits.

* END BLOCK.nn - indicates the end of the begin block with compiler-generated name BLOCK.nn.

* STATEMENT NUMBER n - identifies the start of code generated for statement number n in the source listing.

* INTERLANGUAGE PROCEDURE xxx - identifies the start of encompassing procedure xxx (see chapter 13).

* END INTERLANGUAGE PROCEDURE xxx - identifies the end of encompassing procedure xxx.

* COMPILER GENERATED SUBROUTINE xxx - indicates the start of compiler-generated subroutine xxx.

* END OF COMPILER GENERATED SUBROUTINE - indicates the end of the compiler-generated subroutine.

* ON UNIT BLOCK - indicates the start of an on-unit block.

* ON UNIT BLOCK END - indicates the end of the on-unit block.

* END PROGRAM - indicates the end of the external procedure.

* INITIALIZATION CODE FOR OPTIMIZED LOOP FOLLOWS - indicates that some of the following code has been moved from within a loop by the optimization process.

* CODE MOVED FROM STATEMENT NUMBER n - indicates object code moved by optimization to a different part of the program and gives the number of the statement from which it originated.

* CALCULATION OF COMMONED EXPRESSION FOLLOWS - indicates that the value of an expression used more than once in the program is calculated at the point indicated.

* METHOD OR ORDER OF CALCULATING EXPRESSIONS CHANGED - indicates that the order of the code following has been changed to optimize the object code.

In certain cases, mnemonics are used to identify the type of operand in an instruction, and, where applicable, this is followed by a source-program identifier. The following prefixes are used:

A..	Address constant.
ADD..	Aggregate descriptor descriptor.
BASE..	Base address of a variable.
BLOCK.nn	Label created for an otherwise unlabeled block.
CL.nn	Compiler-generated label.
D..	Descriptor.
DED..	Data element descriptor.
WSP.n	Workspace, followed by decimal number of the block of allocated workspace.
L..	Length of variable.
LOCATOR..	Locator.
RKD..	Record or key descriptor.
VO..	Virtual origin (the address where element 0 would be held for a one-dimensional array, element 0,0 for a two-dimensional array, etc.).

Static Internal Control Section

The static internal control section contains the majority of items that are not executable instructions. The contents of a typical static control section are shown in figure 2.3.

The first part of the static internal control section contains addresses. These are held in the order:

1. Addresses of library modules
2. Addresses of entry points
3. Addresses of label constants that may be assigned to label variables
4. Addresses of external procedures (other than library modules)

The address section is followed by a section known as the constants pool. This contains the following items (if required by the program):

Constants	Constant values used by compiled code.
ONCBs	Control blocks used in error handling. (See chapter 7.)
Descriptors, locators and DEDs (data element descriptors)	Control information used by compiled code and library. (See chapter 4.)
Symbol table address vector and symbol tables	Control information used in data-directed I/O. (See chapter 4.)
Diagnostic statement table	Information on statement numbers.

Items are arranged according to their alignment requirements, those requiring doubleword alignment first, followed by fullword, halfword, byte, and bit.

The final section of the static internal control section holds the static variables. These are held in size order, smallest first: first the variables of 8 bytes or less, next the variables of 2048 bytes or less, and finally any variable greater than 2048 bytes. This system ensures that the smallest possible number of items will require indirect addressing, since it will always be the largest variables that overflow the 4K boundary. Within each division, items are grouped according to alignment stringencies, starting with those requiring doubleword alignment. This method ensures optimum use of storage.

Program Control Section

The program control section contains the executable instructions that are a translation of the PL/I source program. The format of each program control section depends on the contents of the source program. The discussion that follows covers items that will be common to all source programs.

To keep discussions of subjects as complete as possible the chapter also includes descriptions of certain library functions when they are closely allied with the subject under discussion.

	Dedicated registers	Work registers (plus special use)	Preferred registers	Notes
0		General		Cannot be used as base
1		General + address of parameter list		
2	Address of program base			Saved during in-line record I/O and TRT instructions
3	Address of static base			
4		Address of temporary base, if DSA is larger than 3896 bytes		
5		General + static chainback on entry to procedure	Preferred register for DO loop control variable	
6		General		
7		General		
8		General		
9		General		
10		General	Preferred registers for DO loop control when BXLE instruction is used	
11		General		
12	Address of TCA			
13	Address of current DSA			
14		General + branch-and-link to library and other routines		
15				

Figure 2.5. Register usage in compiled code

Register Usage

Details of register usage during the execution of compiled code are given in figure 2.5.

Four general registers are used as bases for addressing various types of data; these are known as dedicated registers. The remainder of the registers are used as they are required and are known as work registers

Dedicated registers are:

R2 Program base.

R3 Static base.

R12 TCA pointer.

R13 DSA pointer.

This arrangement of dedicated registers allows compiled code the use of six even/odd work register pairs. These are (0,1), (4,5), (6,7), (8,9), (10,11), and (14,15).

Certain registers have special tasks for

which they are always used, or for which they are preferred and used when available. These tasks are shown in figure 2.5.

Dedicated Registers

Register 2 - Program Base Register: Register 2 is the program base register and is used for branching within the code. When the code exceeds 4K, register 2 is updated so that all branching is done on this register. During in-line I/O (when data management calls are handled by compiled code rather than by library subroutines), and during the execution of TRT instructions, the program base register contents are saved and the register used for other purposes.

Register 3 - Static Base Register: Register 3 points to the start of the static internal control section. The items to be found in this control section in any particular program are listed in the static-storage map put out by the compiler. (See "Static Internal Control Section," later in this chapter.) When the static control section is larger than 4K bytes, a further base register is used.

Register 12 - TCA: Offsets from register 12 are used to address the various fields in the TCA. The TCA is discussed further in chapter 5 and appendix B.

Register 13 - Current DSA: Register 13 points to the current DSA and is used to address the automatic variables declared in the current procedure or block. References to offsets from register 13 which do not appear as names in the assembler language listing are references to the housekeeping fields held in every DSA or to temporaries. These are discussed in chapter 6; a map of the housekeeping information in a DSA is given in appendix B.

Work Registers

Special or preferred uses for work registers are shown in figure 2.5. Special uses are those for which the register is freed and always used. Preferred uses are those for which the register is used when possible.

Floating-Point Registers

Floating-point registers are all used as general work registers for floating-point data.

Library Register Usage

Register usage in library modules is different from that in compiled code. It is shown in figure 2.6.

Register	Usage
1	Work register
2	Work register
3	Program base register (dedicated)
4	Work register
5	Work register
6	Work register
7	Work register
8	Work register
9	Work register
10	Work register
11	Work register
12	TCA pointer (dedicated in both library and compiled code)
13	DSA pointer
14	Work register (always used for branch-and-link to other routines)
15	Work register (used with register 14 for branch-and-link)

Figure 2.6. Library register usage

Two further points about library register usage are worth noting:

1. Registers 14 through 4 are normally saved by the library. This is because the majority of library subroutines use only these registers. Consequently, time can be saved by reducing save-restore requirements. However, some library routines also save one or more of registers 5 through 11.

2. The majority of library subroutines require argument lists that are addressed by register 1. However, certain library routines have their parameters/arguments passed directly in registers. The registers used for this purpose are 1, 5, 6, and 7.

Handling and Addressing Variables

HANDLING AUTOMATIC VARIABLES

Automatic variables have storage allocated on a procedure or begin-block basis. Variables whose length is known during compilation have storage allocated within the DSA of the block in which they are declared. Variables whose length is not known until execution time have their storage allocated in variable data areas (VDAs). VDAs are held in the last-in/first-out storage stack and are acquired in the prologue code after the DSA has been acquired. The same method is used as is used for acquiring the DSA (see above under "Prologue Code.")

Automatic variables when used in the block in which they are declared are addressed from register 13, if they are held in the DSA. If they are held in a VDA, a separate base is set up for the VDA and they are addressed from this.

Within a DSA, automatic variables are held in size order. First those of 8 bytes or less, then those of 2048 bytes or less, and finally those larger than 2048 bytes. Within each group items are held in alignment stringency starting with items that require doubleword alignment. This arrangement results in the minimum number of variables overflowing the 4096 byte addressing boundary. The contents of a typical compiled code DSA are shown in figure 2.7.

Automatic variables known in any procedure or block are those that are declared in that procedure or block, or in any encompassing procedures or blocks. The method used to address automatic variables in outer blocks is as follows. The address of the DSA of the block in which the required variable was declared is placed in the current DSA. This address can then be accessed from register 13. This is done in the prologue. (Frequently, the value is retained in the register used in the initial load and not reloaded when the variable is accessed.) Typical code would be

```
L   7,96(0,13)    Pick up address of
                   correct DSA
L   8,108(7)      Place value of variable
                   in register 8
```

COMPILER-GENERATED TEMPORARIES

Because PL/I statements can contain an unlimited number of operands, it is frequently necessary to set up fields containing intermediate results. These fields are known as temporary variables (temporaries) and are allocated within the DSA of the associated block, provided that the size of storage required is known at compile time. To simplify addressing the temporaries, register 4 is used to point at the start of the area used for storing them, if the DSA requires more than 30896 bytes of storage.

Because temporary storage is continually being reused, the same storage area will not always hold the same temporary.

Temporaries for Adjustable Variables

Where a temporary is needed to hold a value for an adjustable variable, its size is not predictable until execution. In such cases, a VDA is acquired for the temporary value.

CONTROLLED VARIABLES

Controlled variables are addressed through a field that holds the address of the most recent allocation of the variable. For internal controlled variables, this address is held in the static internal control section. For external controlled variables, a separate control section is generated. When no allocations of the controlled variable have been made, the address field is set to zero.

Each allocation of a controlled variable holds the address of the previous allocation in a chainback field at its head. For the first allocation, the chainback field is set to zero.

The stacking and unstacking of controlled variables is handled by the library module IBMBPAF. This, in turn, makes use of IBMDPGR to actually allocate or free the storage for the variables.

Control Block

The control block area at the head of each controlled variable is four words in length and consists of the following fields.

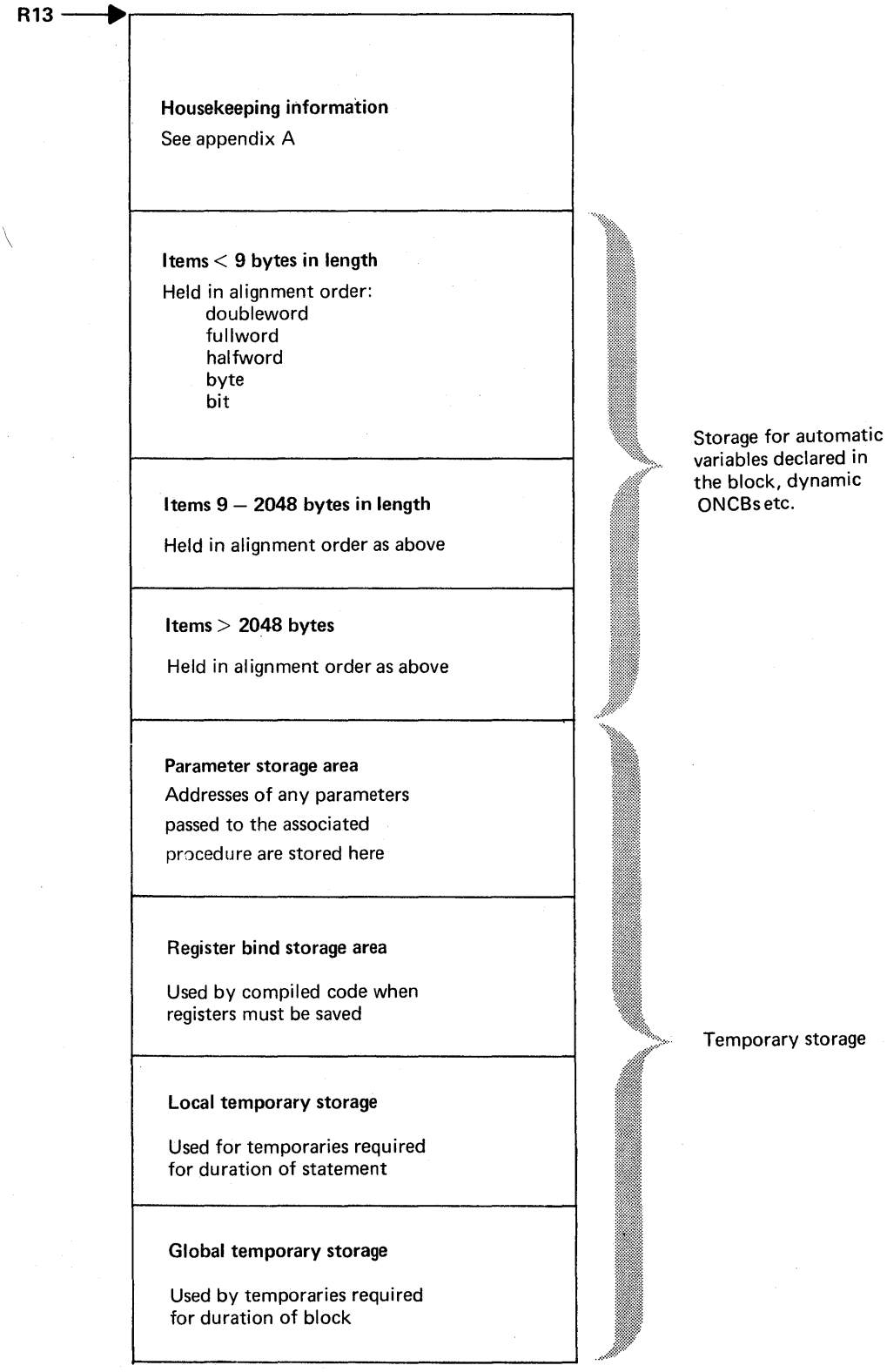


Figure 2.7. Typical contents of a compiled code DSA

Word 1 The first word is used for chaining.

Word 2 Word 2 contains the length of the variable, including the control block.

Word 3 Word 3 points to the address of the previously allocated variable, or contains zero if there is none. (This address is word 5 of the area used by the previous allocation, because the address is that of the start of the variable itself rather than the control block.)

Word 4 Word 4 is unused.

Allocating a Variable

A controlled variable is allocated when IBMBPAF is entered by entry point A. The length of the storage required and the address of the anchor word which will hold the address of the current invocation are passed in register 1. The length is increased by 16 bytes to allow for the control block, and the module IBMDPGR called to allocate the storage. The control block is then initialized. The first bit in the anchor word is set to indicate that the controlled variable has been allocated, the old address in the anchor word having been set in the chainback field (word 3) in the control block at the start of the variable. If there have been no previous allocations, this address is zero.

Freeing a Controlled Variable

Freeing a controlled variable is carried out by IBMBPAF when entered via entry point B. The address of the anchor word is passed by compiled code. If the chainback field in the variable is zero, the first bit of the anchor word is set on, and the area freed by calling IBMDPGR.

If the chainback field is not zero, the address in the chainback field is placed in the anchor word, and IBMDPGR called to free the area.

BASED VARIABLES

Based variables are addressed by using the contents of the pointer on which they are based. The pointer is addressed in the usual manner, depending on its storage class.

When a based variable is allocated, a call to the storage management module IBMDPGR is made. IBMDPGR acquires storage in the non-LIFO dynamic storage area and returns the address of the storage in register 1. The address held in register 1 is then placed in the pointer on which the allocated variable is based.

When the variable is freed, a further call to IBMDPGR is made to free the storage. (Details of the functions of IBMDPGR are given in chapter 6.)

Pointers: Pointers are held as fullwords. The null pointer value is X'FF000000'.

STATIC VARIABLES

Static internal variables are held in the static internal control section and are addressed from register 3.

Static external variables are held in separate control sections and are addressed from an address constant in the static internal control section.

ADDRESSING BEYOND THE 4K LIMIT

As described above, variables and temporaries can, in the simplest case, be addressed by using an offset from one of the base registers. However, as the space required for any particular type of storage can exceed the maximum offset allowed in addressing (4096 bytes), it is necessary to have a scheme to allow addressing of variables beyond this limit.

The method used is to divide storage for automatic variables, temporaries, and static variables into sections of 4096 bytes. The addresses of the second and subsequent sections are then placed in the first section. Addressing of an automatic variable beyond the 4096-byte limit is typically done by code resembling the following:

```
L  6,92(0,13)    Place address of 4K
                  boundary in register 6.

AH 7,96(0,6)     Address variable by using
                  offset from 4K boundary
                  placed in register set
                  up in last instruction.
```

A similar system is used for addressing any static variables and temporaries which

are at an offset greater than 4096 bytes. The addresses are held in the following areas:

Automatic	Immediately following the housekeeping information of the DSA.
Static	At the head of the first section of static storage.
Temporaries	At the head of temporary storage, following bases of parameters, register save area, and addresses of any outer DSAs.

Constants and variables are held in order of size, with the smallest first. This minimizes the number of items that overflow the 4K boundary.

Handling Data Aggregates

PL/I data aggregates are structures and arrays. This includes both arrays of structures and structures of arrays.

Array elements are addressed from the virtual origin of an array. This is the point at which the element whose subscripts are all zeros is held, or would be held if no such element is included in the array. Each element can be accessed by using a multiplier for each dimension. The multiplier is the distance between elements in a cross-section of an array. For example, in an array B(9,9) the multiplier for the first dimension is the distance between elements B(1,1) and B(2,1); the multiplier for the second dimension is the distance between elements B(1,1) and B(1,2).

If the bounds of the array and the length of the elements of the array are known during compilation, the values of multipliers can be calculated and placed as constants in the static internal control section. For accessing an element with a constant subscript, the offset from the virtual origin can be calculated during compilation. If the subscript value is a variable, the multiplier must be picked up from static storage during execution and the value calculated.

If the bounds or extents of an array are not known during compilation, a control block known as an array descriptor is set up. This control block is used to hold necessary information about bounds, multipliers, etc. The information is placed in the control block during execution. Array descriptors are described in chapter 4.

Structures are treated in a similar manner. Where all information about a structure is known, it is mapped during compilation and offsets to each item from the start of the structure are known to compiled code. If a structure cannot be mapped during compilation, it is mapped during execution, and the offsets within the structure are placed in a control block known as a structure descriptor. To access an item in the structure, compiled code finds the offsets and calculates the address of each element from them. Structure descriptors and the process of mapping during execution are described in chapter 4.

ARRAYS OF STRUCTURES AND STRUCTURES OF ARRAYS

Arrays of structures and structures of arrays are held as they are declared.

The array of structures

```
S(2),
     B,
     C;
```

would be held in the order S(1).B, S(1).C, S(2).B, S(2).C.

B and C are known as interleaved arrays, because the elements within each array are not contiguous.

The structure of arrays

```
S,
 B(2)
 C(2);
```

would be held in the order S.B(1), S.B(2), S.C(1), S.C(2).

Elements are accessed as array elements in both cases. In the array of structures shown above, both B and C are treated as separate arrays with their own virtual origins and multipliers. When possible, the values of multipliers are calculated during compilation. When adjustable bounds or extents are involved, the necessary data for both arrays of structures and structures of arrays is placed in a structure descriptor (see chapter 4).

ARRAY AND STRUCTURE ASSIGNMENTS

Assignments between structures and arrays of the same format are done by MVC instructions. Provided an array is not interleaved, an assignment will be made to it as a whole, and the elements will not be moved one at a time. Similarly, structures that are contiguous and have the same format are moved as a whole.

Handling Flow of Control

In PL/I, five types of statement can result in non-consecutive flow of control. These statements are:

CALL statements

END statements

RETURN statements

Function references

GOTO statements

The first four of these are concerned with the block structure of the PL/I program and involve passing control from one block to another. GOTO statements can result in branches to code that is either in the current block, or in any other active block.

Consecutive flow of control also ceases when an error or program interrupt occurs. The methods used to handle error and PL/I condition situations are described in chapter 7, "Error Handling."

ACTIVATING AND TERMINATING BLOCKS

CALL, END, and RETURN statements, and function references, all result in the activation or termination of blocks. The block structure of PL/I, as explained in chapter 1, is implemented by means of a hierarchy of DSAs.

Each block (begin block, procedure block, or on-unit block) executes on its own program base that is set up at the end of the prologue code for each block. This base is marked in the object code listing with:

* PROCEDURE BASE

In the PL/I optimizing compiler, blocks are always called by means of a BALR instruction on registers 14 and 15. Within the prologue code, the registers are stored in the DSA of the calling block, and a new DSA is set up to hold the automatic variables of the new block plus a certain amount of environmental information such as the enablement or disablement of certain conditions.

When a block is terminated, the registers of the calling block are restored, and a branch is made on register 14. This immediately returns control to the instruction after the BALR issued in the preceding block. The DSA of the called block is automatically discarded because all fields in the DSA, including the pointer to the next available byte of free storage, were addressed from register 13. Because register 13 has been altered, the values that apply to the calling block automatically become current when the calling block's registers are restored.

PROLOGUE AND EPILOGUE CODE

Every PL/I begin block or procedure block has prologue and epilogue code. The prologue prepares the environment for the associated block and acquires storage for automatic variables, compiler-generated temporaries, and workspace. The epilogue frees the storage acquired for the block, restores the environment of the calling block, and returns control to the calling block.

Prologue

The prologue appears on the object-program listing between REAL ENTRY and either PROCEDURE BASE or BLOCK BASE. Every prologue has to acquire a dynamic save area (DSA) for the new block. (The DSA is a register save area concatenated with housekeeping information, plus storage for automatic variables and temporaries.) Other jobs that may be done in the prologue code are:

- Initialization of automatic variables that have the INITIAL attribute.
- Initialization of pointers and locators that have the INITIAL attribute.
- Movement of parameter addresses passed to the procedure to the correct location.
- Acquisition of storage for adjustable variables.

```
| STM       14,12,12(13)          Store registers of calling program.
| BC        *+16                  Branch around constants.
| DC        A(STMT NO TABLE)      Constant - address of statement number table.
| DC        F'272'                Constant - length required for new DSA.
| DC        A(STATIC CSECT)       Constant - address of static internal CSECT filled in
|                                 by linkage editor.
| L         3,16(0,15)            Set up R3 as static base.
| L         1,76(0,13)            Set R1 to old NAB (start of new DSA).
| L         0,12(0,15)            Place length required for new DSA in R0.
| ALR       0,1                   Add old NAB (in R1) and length required for DSA (in
|                                 R0).
| CL        0,12(0,12)            Compare with EOS in TCA.
| BNH       *+10                  Branch around library call if new DSA fits segment.
| L         15,116(0,12)          Load address of stack overflow routine (IBMBPGRC) from
|                                 TCA.
| BALR      14,15                 Branch to overflow routine.
| L         14,72(0,3)            Pick up library workspace address.
| LR        15,0                  Place NAB address in R15.
| STM       14,0,72(1)            Store library workspace address and current and
|                                 end-of-prologue NAB addresses in new DSA.
| ST        13,4(0,1)             Set up backchain to previous DSA.
| LA        13,0(1,0)             Point R13 at new DSA.
| ST        5,88(0,13)            Set up static backchain.
| MVI       0(13),X'80'           Set up housekeeping flags - see appendix B.
| MVI       1(13),X'00'           
| MVC       84(4,13),166(3)       Set up enable cells - see chapter 7.
|
| Other code as required          Other tasks may be carried out at this point.  (Such
|                                 as, initialization of variables with the initial
|                                 attribute, acquiring a VDA for adjustable variables,
|                                 and setting up certain error-handling fields.)
| BALR      2,0                   Set R2 as program base.
```

Figure 2.8. Typical prologue code

```
| L    13,4(0,13)       Chainback
| LM   14,12,12(13)     Restore registers of
|                       preceding block
| BR   14               Return
```

Figure 2.9. Epilogue code

- Initialization of certain items for argument lists.
- Setting-up certain interrupt-handling information such as ONCBs and enable cells. (See chapter 7.)

An example of prologue code is shown in figure 2.8.

Two backchains are set up. The dynamic backchain, which points to the DSA of the calling or preceding block, and the static backchain, which points to the DSA of the statically encompassing block. For the main procedure, the dynamic backchain points to the dummy DSA, and the static backchain is set to zero. The address of the statically encompassing block is passed in register 5.

Static backchains are used in tracing the scope of names and the enablement of PL/I conditions.

For PL/I procedures with COBOL or FORTRAN in the OPTIONS option, the prologue is considerably different. See chapter 13, "Interlanguage Communication."

The format of the DSA is shown in figure 2.7.

Epilogue

Epilogue code comprises the instructions generated for END or RETURN statements. These instructions restore the registers to the values that were held when the current block was called. The register values are those stored in the previous DSA. Typical epilogue code is shown in figure 2.9.

For the external procedure the epilogue code is slightly different. The address of the current DSA is saved in register 0, and

return made by a BALR instruction using registers 1 and 14. This allows return to the program if a FINISH on-unit has to be executed.

CALL Statements

CALL statements are executed by picking up the address of the block to be called from static storage. A BALR instruction is then carried out on registers 14 and 15. If arguments are being passed to the called procedure, an argument list is set up in temporary storage, the first bit of the last argument is set to '1', and register 1 is pointed at the argument list. Typical code would be:

```
00031A  18 5D          LR   5,13
                       Load static backchain
                       address
00031C  58 F0 3 020    L    15,A...X
                       Pick up address of
                       procedure X
000320  05 EF          BALR 14,15
                       Branch to procedure
```

Function References

Function references are compiled in exactly the same way as CALL statements. If the function returns a value, an extra field is placed as the last argument in the list. The returned value is placed in this field when the function is completed. Typical code would be:

```
0001FE  41 90 6 0B4    LA   9,B
000202  50 90 3 0BC    ST   9,188(0,3)
000206  41 90 6 0B0    LA   9,A
00020A  50 90 3 0C0    ST   9,192(0,3)
                       Set up parameter list
00020E  18 5D          LR   5,13
                       Load static backchain
                       address
000210  41 10 3 0BC    LA   1,188(0,3)
                       Point register 1
                       at parameter list
000214  58 F0 3 008    L    15,A...DOUBLE
                       Place address of
                       function
                       (DOUBLE) in R15
000218  05 EF          BALR 14,15
                       Branch to function
```

END Statement

END statements result basically in restoring the registers of the calling block and branching to the value held in register 14 of that block.

Code compiled for an END statement of an internal block takes the following form:

```
000402  58 D0 D 004    L    13,4(0,13)
                       Pick up DSA backchain
000406  98 EC D 00C    LM   14,12,12(13)
                       Restore registers
00040A  07 FE          BR   14
                       Branch to procedure
```

For main procedures, certain further actions have to be taken. Because the end of a main procedure raises the FINISH condition, it is necessary to save the current value of register 13 so that the error handler may search the DSA chain for a FINISH on-unit. As it is possible to request a SNAP trace in a FINISH on-unit, it is also necessary to save the address of the END statement. For this reason, the branch is made with a BALR instruction rather than a branch instruction as used for internal blocks. Typical code would be:

```
00188C  18 0D          LR   0,13
                       Save current DSA
                       address in R0
00188E  58 D0 D 004    L    13,4(0,13)
                       Pick up DSA backchain
001892  58 E0 D 00C    L    14,12(0,13)
                       Restore register 14
001896  98 2C D 01C    LM   2,12,28(13)
                       Restore registers 2
                       through 12
00189A  05 1E          BALR 1,14
                       Branch to initializa-
                       tion routine saving
                       branch address in
                       register 1
```

RETURN Statement

RETURN statements are executed in a similar way to END statements, but result in the termination of a procedure rather than a block. Consequently, before the restoration of the registers, a chainback must be made to the correct DSA. A chainback is made through any begin blocks. The depth of nesting can be determined during compilation, so the backchain can be loaded the required number of times before the branch is made. Typical code would be:

```
0003EC  58 D0 D 004    L    13,4(0,13)
0003F0  58 D0 D 004    L    13,4(0,13)
                       Pick up DSA backchain
0003F4  98 EC D 00C    LM   14,12,12(13)
                       Restore registers
0003F8  07 FE          BR   14
                       Branch to procedure
```

Note: If the procedure in which the RETURN statement occurs is a main procedure, the code will take the form compiled for an END statement for an external procedure (see above).

GOTO STATEMENTS

The implications of a GOTO statement depend on whether the label branched to is within the block or external to it. If the label is outside the block, the branch implies that one or more blocks must be terminated. If the label in the GOTO statement is a label variable, it is not always possible to determine during compilation whether the label will be in the same block as the GOTO statement. Consequently, interpretive code is used for label variables.

For GOTO statements to a label constant within the block, the compiler produces a straightforward branch instruction. For GOTO statements that may pass control to another block, compiled code calls the interpretive code in the TCA.

Interpretive code to handle a GOTO out of block is held in the TCA. To implement a GOTO that will or may transfer control out of the block, compiled code branches to code in the TCA. The code in the TCA checks to see whether it is one of a small number of special cases, and, if it is, calls a subroutine of the PL/I resident library module IBMDPIR. IBMDPIR is the program initialization routine and is always link-edited. In other circumstances, the GOTO code in the TCA handles the branch and any block termination involved.

The special cases all occur when executing code handled by library modules. These library modules flag the TCA and their own DSA to indicate that special action must be taken when a GOTO occurs.

To execute a GOTO statement, three things must be known:

1. The address of the instruction to be branched to.
2. The address of the program base.
3. The address of the DSA associated with the instruction.

The label constant holds items 1 and 2. Item three is held in a label variable. (For formats see appendix B). When a branch is made to a label constant using the GOTO code in the TCA, a label temporary is created. The label temporary has the same format as a label variable, consisting of the address of the label constant followed by the address of the associated DSA.

GOTO Within a Block

The optimizing compiler produces code that assumes that the registers retained across the execution of a labeled statement will be 2, 3, 12, and 13. These are the program base, the static base, the address of the TCA, and the address of the current DSA. All other register values may be different when control pases through the labeled statement on different occasions.

The enablement of conditions may differ in the GOTO statement and in the labeled statement. Within a block, the enablement status may be varied only for the duration of a single statement. The GOTO therefore resets the block enablement status before the branch is taken. If the labeled statement has a different enablement status from the block, it will be automatically reset in the labeled statement.

As explained in chapter 7, "Error and Condition Handling," the enablement of conditions is recorded by enable cells. Two sets are used: the block enable cells retain the enablement situation at the start of the block, which can consequently be restored at any time; the current enable cells hold the enablement situation that is current, which, as explained earlier, may differ from that at the start of the block.

A GOTO within block normally takes the form of a simple branch instruction plus any alteration of the enablement bits that may be necessary to reset the enablement situation to that at the start of the block. Typical code would be:

```
000F1A  47 F0 2 0C8    B    INPUT
                       Branch to correct
                       address in compiled
                       code (label name is
                       "INPUT")
```

The optimizing compiler attempts to retain the same block base for all branches within a block. However, this is not always possible and, if the code for the block is longer than 4096 bytes, it may be necessary to set up a new base when a GOTO statement is executed. As all labels are stored with both their address and their base, this presents no problem. The address of the label and the value of its base form the value of the label constant. The value of the base is placed in register 2, and a branch is made to the label address.

When a GOTO to a label within the block is made, there is no need to reset registers 3, 12, or 13 as these are not altered within a block.

Labeled statements within a block have an effect on optimization in that, apart from the bases and block addresses mentioned above, values cannot be retained in registers beyond a labeled statement.

GOTO Out of Block

GOTO statements that transfer control from a block have to overcome the problems described above, plus problems of block termination.

For a GOTO out of block or to a label variable, compiled code makes a call to the GOTO code in the TCA, which is held at offset 128(decimal). The GOTO code receives, through registers 14 and 15, either the contents of the label variable or the equivalent information for a label constant, namely the address at which the label constant is held, and the address of the DSA of the block in which the label appears.

The GOTO code first tests to see if a change of block is being made. If not, the enablement is reset as described below. If a change of block is being made, then, if FLOW or COUNT is in effect a call is made to IBMDEFL to update the flow or count tables. Next, the GOTO subroutine of IBMDPIR is called to determine if a valid GOTO is being undertaken. The GOTO subroutine ensures that the target block for the GOTO is still active and is not an invalid address. Provided that this is not an abnormal GOTO, registers 3 and 4 are restored from the target DSA, register 2 is loaded from the second word of the label constant, and register 13 is set to the address of the target DSA. The routine then branches to the appropriate point in code which is picked up from the address of the label constant, passed in register 14.

The enablement situation at the start of the block has to be restored, and this is done by setting the current enable cells in the DSA to the value of the block enable cells. If the current enable cells indicate that CHECK is enabled, the module IBMDPGD is called. A search is made for a qualified CHECK ONCB, so that the enable cells may be set to the start-of-block situation in this ONCB.

In a similar manner, it may be necessary to restore the NAB value to that at the start of the block. This will be necessary if the statement that invoked the block acquired a VDA. The start-of-block NAB value is retained in the DSA and is known as the end-of-prologue NAB. If a VDA has been acquired, the fact is flagged in the flag byte of the DSA, and the GOTO code places the end-of-prologue NAB value in the current NAB field.

Such action is never required within a block, as VDAs are only acquired for the duration of one statement and are never used for GOTO statements. Typical code would be:

```
     GOTO label-constant (out of block)

000226  18 F6          LR    15,6
                       Place address of DSA
                       in R15
000228  41 E0 3 088    LA    14,136(0,3)
                       Place address of
                       label constant in R14
00022C  47 F0 C 080    B     128(0,12)
                       Branch to GOTO code
                       in TCA
```

GOTO Label Variable

GOTO label variable statements are treated in different ways depending on whether optimization has been specified.

For NOOPTIMIZE, they are all treated as GOTO out of block; for OPTIMIZE(TIME), a check is made to determine whether they could be out-of-block branches. The check is made by testing a label list, which is a list of the label constants to which the label variable may be assigned. If the programmer has supplied a label list, it is used. Otherwise, a list is generated containing all the label constants that are assigned to label variables. If a branch to any of the labels in the list could result in a GOTO out of block, all GOTO statements referring to the label variable are treated as GOTO out-of-block situations. Typical code would be:

```
     GOTO label-variable

0000D0  98 EF D 0A8    LM    14,15,168(13)
                       Load R14 and R15 with
                       label variable
0000D4  47 F0 0 080    B     128(0,12)
                       Branch to GOTO code
                       in TCA
```

GOTO Only On-Units

On-units containing only a GOTO statement

are not compiled as separate program blocks. Instead the ON control block (ONCB) normally used to address the on-unit, is specially flagged and, instead of containing the on-unit address, contains the offset within the associated DSA of a word which contains the address of a label variable or label temporary. This variable or temporary contains the address of the label constant to which control is to be transferred and the DSA associated with the label constant.

Before an on-unit is entered, the error handling module, IBMDERR, inspects the ONCB and, for a GOTO only on-unit, transfers control by loading registers 14 and 15 with the label variable or temporary and passing control to the GOTO code in the TCA.

Interpretive GOTO Subroutine

If the test in the GOTO subroutine in IBMDPIR indicates that an abnormal GOTO may occur, control passes to the interpretive GOTO subroutine. This routine is held as another subroutine of the initialization routine IBMDPIR, and is consequently always link-edited.

All the situations that can lead to abnormal GOTOs are handled by library routines. When these routines are entered they flag the TCA and their own DSA to indicate that an abnormal GOTO may occur.

The interpretive GOTO subroutine chains back through the DSA's and when it finds the flagged DSA passes control to the associated module which does any necessary housekeeping.

The special cases in which the interpretive GOTO subroutine is called are:

GOTO out of a SORT E35 or E15 routine.

GOTO out of an EVENT I/O on-unit.

GOTO out of an on-unit which results in the termination of a COBOL or FORTRAN routine.

These situations are covered more fully in the relevant sections of this publication (See index).

The interpretive GOTO subroutine is described in DOS PL/I Resident Library Program Logic manual.

Argument and Parameter Lists

In PL/I usage, a parameter list is a list of the items a program expects to be passed; an argument list is a list of the items that are passed by the calling routine.

Between PL/I routines, addresses are always passed rather than the arguments themselves. For strings, structures, arrays, and areas, the addresses of locators are passed rather than the addresses of the arguments themselves. The format of locators and the reasons for their use are given in chapter 4.

When arguments are passed to routines whose entry points are declared with the ASSEMBLER, COBOL, or FORTRAN attribute, the address of the data itself must be passed. The method used is described in chapter 13, "Interlanguage Communication."

Arguments are passed in an argument list addressed by register 1. Normally the list is set up in static storage. The addresses are loaded into consecutive registers and placed in the list by an STM instruction. If the procedure is reentrant or recursive, the list is moved into the temporary storage area of the DSA before the call is made.

The addresses passed in the argument list are moved into the parameter storage area, which is held at the head of temporary storage and is addressed by register 4. (See figure 2.9.) Parameters are then accessed by picking up the addresses from this area.

Dummy arguments, when they are required, are set up by the calling program. Consequently, the called program can treat all arguments in the same manner.

LIBRARY CALLS

Library calls are a feature of every object program. All library calls that appear in the object-program listing are to modules in the resident library. Transient library routines are called by bootstrap routines which are held in the resident library.

The number of library calls used depends on the source program and the level of optimization specified. For OPTIMIZE (TIME), the minimum number of library calls will be made. If NOOPTIMIZE is specified, library calls will be made where this will speed compilation. The standard default is NOOPTIMIZE.

```
LA    1,40(0,4)          Point R1 at argument
                         list
LA    14,VO..U(11)       Load address of
                         argument in register
LA    15,DED..VO..       Load address of
      U(11)              argument in register
STM   14,15,0(1)         Store into argument
                         list
L     15,A..IBMBSLOA     Pick up address of
                         routine from static
                         internal control
                         section and place in
                         R15
BALR  14,15              Branch and link to
                         routine
```

Example 1. Call to library routine that has been link-edited and whose address is held in the static internal control section. The arguments passed are addressed by register 1.

```
L     15,116(0,12)       Load address of
                         routine held in TCA
BALR  14,15              Branch and link to
                         routine
```

Example 2. Call to library routine whose address is held in the TCA

Figure 2.10. Examples of library calling sequences

Figure 2.11 shows examples of sequences used for calling library modules. The majority of library calls can easily be recognized by the appearance in the listing of the letters "IBM" followed by five letters specifying the module name and entry point. To call a module, its address is loaded into register 15, and a BALR instruction is carried out on registers 14 and 15.

The fifth letter of the entry point name is mnemonic of the type of module that is being called. Figure 2.12 gives the meaning of the mnemonics. Full details of the library modules are given in the program product publications DOS PL/I Transient Library: Program Logic and DOS PL/I Resident Library: Program Logic.

A further discussion of library module naming conventions is given chapter 3.

Setting-Up Argument Lists

Before a call is made to a library module, an argument list must normally be set up. This is done in one of several ways, depending on the library module. The majority of library calls require the method shown in figure 2.10, example 1. This consists of loading the list into sequential registers starting at register 14, and then using a store-multiple instruction to place the arguments into an area of static storage, whose address is then loaded into register 1. Argument lists are set up as far as possible during compilation and, where necessary, completed during execution.

Addressing the Subroutine

Library addresses are normally held in static storage and addressed as an offset from register 3. However, the addresses of certain library routines are held in the TCA or the TCA appendage and addressed from register 12. They are addressed either directly or indirectly as shown in example 2 of figure 2.10. The names of these routines do not appear on the listing; however, they can be identified by their offset from the start of the TCA (see figure 2.12).

IBMBA---	Array handling
IBMBB---	String handling
IBMBC---	Conversion
IBMBE---	Error handling
IBMBI---	Interlanguage communication
IBMBJ---	Date/time/delay/wait
IBMBK---	Dump/sort/checkpoint/restart
IBMBM---	Mathematical
IBMBO---	Open/close
IBMBR---	Record I/O
IBMBS---	Stream I/O
IBMBT---	Completion pseudovariable routine

Figure 2.11. Mnenomic letters in library module entry-point names

Where possible, do-loops are carried out by means of a BXLE instruction, because this

Offset from start of TCA (Register 12)		Name of module entry point	Use
Decimal	Hex		
72	48	IBMBPGRD	Stack overflow routine to get VDA
108	6C	IBMBPGRA	Get non-LIFO dynamic storage
112	70	IBMBPGRB	Free non-LIFO dynamic storage
116	74	IBMBPGRC	Stack overflow routine for prologue
120	78	IBMBERRB	Error handler software interrupt
264	108	IBMBJWTA	WAIT module
268	10C	IBMBTOCA	Completion pseudovariable routine
272	110	IBMBTOCB	Event variable assignment routine

Figure 2.12. Offsets where addresses of library modules are held in the TCA

is more efficient than using a simple BCT instruction. BXLE do-loops can be used where the control variable cannot be altered except at the head of the loop, and where it is not subsequently accessed after the completion of the loop. BXLE do-loops cannot be used for the outer of a number of nested do-loops. For outer loops, other branch instructions are used. Code for two of typical nested do-loops is shown below. Note that the code will differ according to the context of the loop.

Source program

```
DO I = 1 to 10;

  DO J = 1 to 10;
       .
       .
       .
       .
       .
     END;

END;
```

Note: For the code to be compiled in the manner shown below, J must not be accessed during the loop, nor after the loop until a new value has been assigned.

Object program

1. Code for outer do-loop

```
      LH        5,596(0,3)     Pick up 1 from
                               constants pool

      STH       5,I            Place 1 in I
CL.1  EQU       *
       .
       .
       .
       .
       .
      LH        5,I
      AH        5,596(0,3)     Increment and
      STH       5,I            store in I
      C         5,598(0,3)     Compare I and
                               constant 10 in
                               static storage
      BNH       CL.1
```

2. Code for inner do-loop

```
      LH        5,596(0,3)     Place 1 in
                               first operand

      LH        10,596(0,3)    Place 1 in
                               second operand
      LH        11,598(0,3)    Place 10 in
                               comparand
CL.2  EQU       *
       .
       .
       .
       .
       .
      BXLE      5,10,CL.2      Increment,
                               test, and
                               branch if
                               necessary.
```

COMPILER-GENERATED SUBROUTINES

The compiler uses internal subroutines to carry out certain functions. These have the advantage over library modules, because they can be tailored for the most common case. When special cases arise, the library routines are called. Compiler-generated subroutines have the further advantage that they are internal to compiled code and consequently need not follow the standard operating system calling sequence.

Compiler-generated subroutines are used for the following purposes.

IELCGIA	Stream I/O input - provides address of source of next edit-directed data or format item
IELCGIB	Stream (edit) I/O input - housekeeping after transmission of data item
IELCGOA	Stream I/O output - provides address of target of next edit-directed data or format item
IELCGOB	Stream I/O output - updates FCB, counts data item, and frees VDA if one was used
IELCGOC	Stream I/O - processes X format items
IELCGMV	Move long (registers 6,7,8,9)
IELCGCL	Compare long (registers 1,6,7,8,9)
IELCGCB	Compare long bits
IELCGON	Dynamic ONCB chaining
IELCGRV	Revert VDA chaining
IELCGBB	Test for '1' bits
IELCGBO	Test for '0' bits

Compiler-generated subroutines are held in separate control sections and are printed at the head of the object-program listing if they are used in a program.

Optimization and Its Effects

Optimization is the attempt to produce the most efficient possible object program. The DOS PL/I Optimizing Compiler adopts a threefold approach:

1. It attempts to compile each statement in the most efficient manner.

2. It modifies the resulting code for each block, in an attempt to make it more efficient (for example, by maintaining values in registers and by using common control blocks for similar items).

3. It examines the source program to discover whether statement flow can be reorganized to produce a more efficient program (for example, by moving code out of loops).

The effect of specifying the compiler option OPTIMIZE (TIME) is that the compiler loads and calls the optimization phases, and executes optimization code in other phases. The optimization phases are described in the publication DOS PL/I Optimizing Compiler: Program Logic.

When NOOPTIMIZE is specified, the optimization phases are not called; no attempt is made to study the flow of the program, and the examination of compiled code for possible improvements is not undertaken on a global basis. More library calls will generally be made if NOOPTIMIZE is specified.

EXAMPLES OF OPTIMIZED CODE

A number of the more noticeable effects of optimization are shown below. These are code sequences which may prove difficult to understand without knowledge of the objectives of optimization. Where possible, the examples of code given are expansions of the examples shown in the language reference manual for this compiler. The examples do not attempt to cover all optimization carried out by the compiler.

Elimination of Common Expressions

This is done by avoiding multiple calculations of the same expression, by holding the value either in temporary storage or in a register. In the examples shown below, the common expression is "B+C". In the first example, the value is held in a register. In the second, it is held in temporary storage, because the value to which it is first assigned is altered. In certain circumstances, the code could be compiled to move the value from the variable to which it was originally assigned to the second variable.

Example 1: Value held in register

Source program

```
2    A=B+C;

3    IF X<Y THEN X=Y;
4    D=B+C;
```

Object program

```
* STATEMENT NUMBER  2
000062  78 00 D 0A4       LE   0,B
000066  7A 00 D 0A8       AE   0,C
00006A  70 00 D 0A0       STE  0,A

* STATEMENT NUMBER  3
00006E  78 60 D 0AC       LE   6,X
000072  79 60 D 0B0       CE   6,Y
000076  47 B0 2 020       BNL  CL.2
00007A  78 60 D 0B0       LE   6,Y
00007E  70 60 D 0AC       STE  6,X
000082              CL.2  EQU  *
```

```
* STATEMENT NUMBER  4

 * CALCULATION OF COMMONED EXPRESSION
 FOLLOWS
000082  70 00 D 0B4       STE  0,D
```

Example 2: Value held in temporary storage

Source program

```
    2          A=B+C;
    3          IF X<Y THEN A=6;
    4          D=B+C;
```

Note: A may be altered before subsequent use of expression.

Object program

```
* STATEMENT NUMBER  2
000062  78 00 D 0A4       LE   0,B
000066  7A 00 D 0A8       AE   0,C
00006A  38 20             LER  2,0
00006C  70 20 D 0A0       STE  2,A

* STATEMENT NUMBER  3
000070  78 60 D 0AC       LE   6,X
000074  79 60 D 0B0       CE   6,Y
000078  47 B0 2 024       BNL  CL.2
00007C  78 20 3 010       LE   2,20(0,3)
000080  70 20 D 0A0       STE  2,A
000084             CL.2   EQU  *

* STATEMENT NUMBER  4

 * CALCULATION OF COMMONED EXPRESSION
 FOLLOWS
00008A  70 00 D 0B4       STE  0,D
```

Movement of Expressions out of Loops

When expressions cannot be altered inside a section of code that may be executed a number of times, the expression is moved out of the loop to a position where it will be executed only once, regardless of the number of times that the loop is executed. The process is known as movement of invariant expressions. The most obvious example is in do-loops. However, the compiler analyzes the source program for other types of loop and also moves code from these.

Example 1 shows code moved from a do-loop. Example 2 shows code moved from a loop that has been detected by the compiler. It should be noted that code moved out of loops frequently involves conversion and is not obvious in the source program.

Example 1: Do-loop

Source program

```
    3          DO I=1 TO N;
    4          J=3;
    5             A(I)=B(I);
    6          END;
    7          END;
```

Object program

```
* STATEMENT NUMBER  3
00005E  48 E0 D 0AA       LH   14,N
000062  48 60 3 010       LH   6,16(0,3)
000066  40 60 D 0A8       STH  6,I
00006A  19 6E             CR   6,14
00006C  47 20 2 036       BH   CL.3

 * INITIALIZATION CODE FOR OPTIMIZED LOOP
 FOLLOWS

* CODE MOVED FROM STATEMENT NUMBER 4
000070  48 80 3 012       LH   8,18(0,3)
000074  40 80 D 0AC       STH  8,J

* CODE MOVED FROM STATEMENT NUMBER 5
000078  48 90 3 014       LH   9,20(0,3)
00007C  18 7E             LR   7,14
00007E  8B 70 0 002       SLA  7,2

* CONTINUATION OF STATEMENT NUMBER 3
000082  18 59             LR   5,9
000084  18 B7             LR   11,7
000086  18 A9             LR   10,9
000088             CL.2   EQU  *

* STATEMENT NUMBER 4

* STATEMENT NUMBER 5
000088  78 25 D 0D4       LE   2,VO..B(5)
00008C  70 25 D 0AC       STE  2,VO..A(5)

* STATEMENT NUMBER 6
000090  87 5A 2 02A       BXLE 5,10,CL.2
000094             CL.3   EQU  *
```

Example 2: Compiler-detected loop

Source program

```
    2          L: IF X>Y THEN GOTO BED;
                  /*LOOP BEGINS*/
    3             J=I-N;
    4          X=X+J;
    5       GO TO L;   /*LOOP ENDS*/
    6       BED:   A=X;
```

Object program

```
* INITIALIZATION CODE FOR OPTIMIZED LOOP
FOLLOWS
```

```
* CODE MOVED FROM STATEMENT NUMBER 3
000066  48 E0 D 0AE       LH    14,I
00006A  4B E0 D 0B0       SH    14,N
00006E  50 E0 4 028       ST    14,40(0,4)

 * CONTINUATION OF STATEMENT NUMBER 1

* STATEMENT NUMBER  2

* STATEMENT LABEL  L
000072  78 00 D 0A0       LE    0,X
000076  79 00 D 0A4       CE    0,Y
00007A  47 20 2 042       BH    BED

* STATEMENT NUMBER  3

 * CALCULATION OF COMMONED EXPRESSION
 FOLLOWS
00007E  58 60 4 028       L     6,40(0,4)
000082  40 60 D 0AC       STH   6,J

* STATEMENT NUMBER  4

* END OF COMMON CODE
000086  50 60 4 030       ST    6,48(0,4)
00008A  48 60 3 020       LH    6,32(0,3)
00008E  40 60 4 030       STH   6,48(0,4)
000092  97 80 4 032       XI    50(4),X'80'
000096  78 60 4 030       LE    6,48(0,4)
00009A  7B 60 3 020       SE    6,32(0,3)
00009E  3A 60             AER   6,0
0000A0  70 60 D 0A0       STE   6,X

* STATEMENT NUMBER  5
0000A4  47 F0 2 00C       B     L

* STATEMENT NUMBER  6

* STATEMENT LABEL  BED
0000A8  70 00 D 0A8       STE   0,A
```

Elimination of Unreachable Statements

If the source program contains statements that can never be executed because they are unconditionally branched around, these statements will be ignored by the compiler.

In the example below, the statements between 5 and 8 can never be reached. Consequently, no code is compiled for these statements, and a compiler diagnostic message is issued to indicate that this is the case.

Example

Source program

```
5  GOTO LABEL;
6     IF A<B THEN
                 IF B<C THEN
                           IF A<X THEN
                             B=B*C;
7     ELSE C=B*C;
8 LABEL:  X=X+1;
```

Object program

```
 * STATEMENT NUMBER  5
00008A  47 F0 2 028       B     LABEL

 * STATEMENT NUMBER  8

 * STATEMENT LABEL  LABEL
00008E  78 60 D 0AC       LE    6,X
000092  7A 60 3 018       AE    6,24
                                (0,3)
000096  70 60 D 0AC       STE   6,X
```

Compiler message reads:

"6,7 STATEMENT MAY NEVER BE EXECUTED. STATEMENT IGNORED."

Simplification of Expressions

Certain expressions are simplified for speedier execution. For example, multiplication is simplified to addition, as in the following example.

Example: Multiplication into addition

Source statement

```
2     X=3*B
```

Object program

```
 * STATEMENT NUMBER  2
000062  78 20 D 0A4       LE    2,B
000066  3A 22             AER   2,2
000068  7A 20 D 0A4       AE    2,B
00006C  70 20 D 0A0       STE   2,X
```

Modification of DO-Loop Control Variables

When the do-loop control variable is used for accessing array elements, it is frequently modified to simplify addressing of the array elements.

If, as in the example below, the elements of the array are four bytes long, it simplifies addressing to increment the loop control variable by 4 rather than by 1. When this is done, the increment becomes the distance between the start of successive array elements. Provided that the original value of the loop control variable is the same as that of the first

```
Source program

   2          DCL C(10) FLOAT DECIMAL (6);
   3          DCL B(10) FLOAT DECIMAL (6);
   4           DO I=1 TO 10;
   5            C(I)=B(I);
   6            END;
Object program

  * STATEMENT NUMBER  4
  000066  48 60 3 010       LH    6,16(0,3)          Pick up 1 from static
  00006A  40 60 D 0A0       STH   6,I                Place in I

  * INITIALIZATION CODE FOR OPTIMIZED LOOP FOLLOWS

  * CODE MOVED FROM STATEMENT NUMBER 5
  00006E  48 E0 3 012       LH    14,18(0,3)         Load "4" into R14 from static
  000072  48 90 3 014       LH    9,20(0,3)          Load "40" into R9 from static

  * CONTINUATION OF STATEMENT NUMBER  4
  000076  18 B9             LR    11,9               Load "40" into R11 for BXLE
  000078  58 A0 3 012       L     10,18(0,3)         Load "4" into R10
  00007C  18 5E             LR    5,14               Load "4" into R5
  00007E             CL.2   EQU   *

  * STATEMENT NUMBER  5
  00007E  78 45 D 0A4       LE    4,VO..B(5)         Pick up VO..B+R5
  000082  70 45 D 0CC       STE   4,VO..C(5)         Place in VO..C+R5

  * STATEMENT NUMBER  6
  000086  87 5A 2 018       BXLE  5,10,CL.2          Increment R5 by 4, test for end of
                                                     loop, and branch or continue
```

Figure 2.13. Code showing modification of do-loop control variable

```
Source program

   2          IF (A=D)|(C=D) THEN
                 X=Y+Z;
Object program

  * STATEMENT NUMBER  2
  000062  78 00 D 0A0       LE    0,A                Pick up A
  000066  79 00 D 0A4       CE    0,D                Compare A and D
  00006A  47 80 2 018       BE    CL.3               Branch if equal
  00006E  78 40 D 0A8       LE    4,C                Pick up C
  000072  79 40 D 0A4       CE    4,D                Compare C and D
  000076  47 70 2 024       BNE   CL.2               Branch if not equal
  00007A             CL.3   EQU   *
  00007A  78 60 D 0B0       LE    6,Y
  00007E  7A 60 D 0B4       AE    6,Z                  X=Y+Z
  000082  70 60 D 0AC       STE   6,X
  000086             CL.2   EQU   *
```

Figure 2.14. Code showing branch around redundant expression

bound of the array, the loop control variable in turn becomes the offset of the element from the virtual origin of the array.

If the loop control variable is altered, this means that the increment and final value must also be altered. Thus the loop in the example below, instead of being incremented from 1 to 10 by 1, is incremented from 4 to 40 by 4. Note that the value of the loop control variable is set at the start of the loop but is not incremented. If the value of the loop variable is required after the loop has been executed, this type of optimization cannot take place.

In the example in figure 2.13, the control variable is held in register 5 using a BXLE instruction. The array elements are addressed by using register 5 as the offset from the virtual origins of arrays C and B. As register 5 starts the loop with the value of 4 and is incremented by 4 for each iteration of the loop, this gives the correct address. Both arrays begin 4 bytes from their virtual origins, and each array element is 4 bytes long.

Branching Around Redundant Expressions

If a series of tests are to be made and action taken if any of the tests proves positive, the compiler takes the requisite action as soon as the first positive test is found.

In the example in figure 2.14, a test is first made to see if A=D. If so, the value of Y+Z is assigned to X without a further test being made to see if C=D. Note that the last test is for inequality, so that if the variables are equal, control will continue with the code that assigns the value to X.

Rationalization of Program Branches

When the length of a program is greater than 4096 bytes and, consequently, it cannot be addressed from one base register, an attempt is made to update the base at the most efficient point, so that there will be as few changes of program base as possible during execution. The aim is to avoid any program branches which move from the scope of one base register to the scope of another.

The program base register is register 2, and this is updated when necessary. As register 2 is required for in-line record I/O and TRT instructions, the program base is saved and restored after such use.

Use of Common Constants and Control Blocks

Constants and control information used more than once are generated only once in static storage. Thus for the statements X=768, Y=768, the constant value 768 will be picked up from the same address in both cases. Similarly, compiler-generated control information, such as DEDs and descriptors (see chapter 4), are generated only once if a number of variables require identical control information.

The process of avoiding duplication is known as commoning. It should be noted that constants may not be commoned if they are not used in the same way. In the following example, constant '123' is stored in a different form for assignment, and exponentiation.

```
Source program

   2          X=123;   /*COMMONED ITEM*/
   3          Y=123*Z;
   4          V=V**123;
   5          A=123;   /*COMMONED ITEM*/
Object program

  * STATEMENT NUMBER  2
  000066  78 00 3 020       LE   0,32(0,3)            /*COMMONED ITEM*/
  00006A  70 00 D 0A0       STE  0,X

  * STATEMENT NUMBER  3
  00006E  78 20 D 0A8       LE   2,Z
  000072  6C 20 3 018       ME   2,24(0,3)
  00007A  70 20 D 0A4       STE  2,Y

  * STATEMENT NUMBER  4
  00007E  41 90 D 0B4       LA   9,V
  000082  50 90 3 024       ST   9,36(0,3)
  000086  50 90 D 02C       ST   9,44(0,3)
  00008A  96 80 3 02C       OI   44(3),X'80'
  00008E  41 10 3 024       LA   1,36(0,3)
  000092  58 F0 3 00C       L    15,A..IBMBMXSA
  000096  05 EF             BALR 14,15

  * STATEMENT NUMBER  5
  0000B8  78 20 3 020       LE   2,32(0,3)
  0000BC  70 20 D 0B0       STE  2,A                  /*COMMONED ITEM*/
```

Figure 2.15. Code showing use of common constant

Chapter 3: The PL/I Libraries

This chapter explains the use of libraries by the DOS PL/I Optimizing Compiler. The topics covered are: when and why library routines are called, why there is both a transient and a resident library, naming conventions, and two implementation topics that cover all library modules: the use of library workspace and the use of weak external references.

The DOS PL/I Optimizing Compiler is designed to be used in conjunction with the DOS PL/I Resident Library and the DOS PL/I Transient Library. These libraries consist of sets of standard subroutines that are used for the majority of interfaces with the system and for those jobs that can be most efficiently done by the use of interpretive subroutines. The main areas where library modules are used are: input/output, error handling, storage management, conversions, mathematical functions, and various string- and array-handling operations.

Use of library routines simplifies compilation by enabling the compiler to set up an argument list and generate a call to a subroutine, rather than compile the complete code. However, library subroutines are less efficient than compiled code, since they must be generalized routines, whereas compiled code can be specially tailored to the particular program being executed. Furthermore, a library call involves the overhead of saving and restoring registers, and may require the setting-up of various additional control blocks to describe the data (see chapter 4). For these reasons, programs that are optimized for time use as few library calls as possible.

The majority of interfaces between compiled code and the operating system are implemented via library routines. This is done mainly for reasons of implementation convenience, as such interfaces are in this way localized and minimized.

Resident and Transient Libraries

The DOS PL/I subroutine library is divided into two separate program products: the DOS PL/I Resident Library (Program Number 5736-LM4) and the DOS PL/I Transient Library (Program Number 5736-LM5). Resident library modules are link-edited with the executable program phase. Transient library modules are loaded into dynamic storage when they are required. When they are no longer needed, the storage is freed and may be overwritten. Resident library routines have the advantage of speed; transient library routines have the advantage of saving space. By using both types of library, it is possible to produce more efficient programs.

Routines in the transient library are: input/output transmitters, open and close modules, error message modules, and PLIDUMP routines. All other library routines are held in the resident library, including a number of bootstrap routines that load and call transient routines.

The DOS PL/I libraries reside in two direct-access libraries. The resident library is on the relocatable library and the transient library on the core-image library.

The internal logic of individual library modules is described in the publications DOS PL/I Resident Library: Program Logic and DOS PL/I Transient Library: Program Logic. However, in such cases as I/O, error handling, and conversion, where compiled code and a hierarchy of library modules are used in implementing certain features of PL/I, the overall logic is described in this publication. Similarly, an overall explanation of storage management and interlanguage communication is given in this publication.

Naming Conventions

Most PL/I library modules have names of seven letters, the first three letters being IBM. This identifies the module as belonging to one of the PL/I libraries. The remaining letters indicate which particular library the module was written for, and the use of the module.

Each resident library module has two names, the control name (which uniquely identifies the module) and the link-edit name (which is used to link edit the module to the appropriate data set. The link edit name is the same as the name of the first entry point). See figure 3.1. The use of two names allows the compiler to call the appropriate module regardless of the actual module available on the system. For example, there are two WAIT modules, one

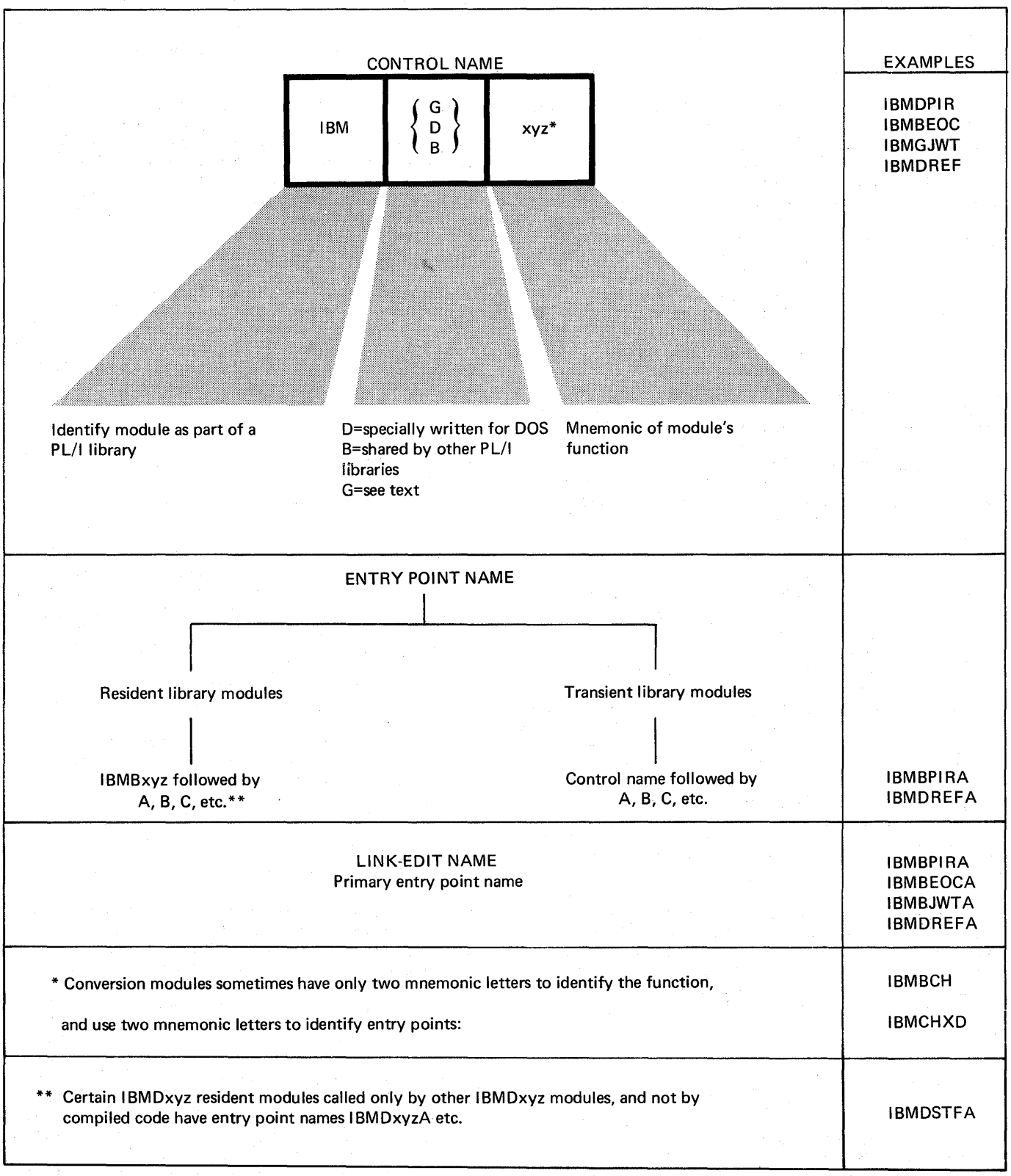


Figure 3.1. Library module names

for machine configurations that support the WAITM macro instruction, and one for those that do not. (The WAITM macro instruction allows waits on multiple events.) These modules have different control names. Machine configurations that support the WAITM instruction will normally have the module with the control name IBMDJWT in their resident library. Machine configurations that do not support the WAITM macro instruction will have the module with the control name IBMGJWT. Both modules have the link-edit name, IBMBJWTA. The compiler can therefore generate appropriate ESD references without knowing which module is available on the system. (Note: Underlinings are not part of the name.) Link-edit names, and all entry point names called from compiled code, have the fourth letter "B".

Resident library module entry points that can be called from compiled code have names in which the fourth letter is "B", regardless of the control name. An additional letter or letters are used to make up the name to 8 letters. Normally the primary entry point is "A" (IBMBJWTA) the second entry point "B" etc. Transient library modules and certain resident library modules not called directly by compiled code have entry point names consisting of the control name plus an additional letter.

Library Workspace

For certain library routines, DSA (dynamic storage areas) are not acquired in the same way as they are for source program subroutines. Instead of the storage being acquired from the LIFO stack, space is allocated, in the program management area, for two pre-formatted DSAs. These DSAs are known as levels of library workspace. Their format can be seen in figure 3.2. Library workspace (LWS), provides a fast method for library routines to obtain DSAs. All the library routines have to do is to address the DSA and set the chainback field. There is no need to test to see if there is enough space for the DSA, and the NAB pointer does not have to be reset, because the next available byte is not changed.

The PL/I libraries have been so designed that two levels of library workspace are the maximum required. This does not mean, however, that more than two modules are never called. Some library modules - for example, the error handler - use DSAs in the LIFO stack for working storage.

FORMAT OF LIBRARY WORKSPACE

Library workspace is designed so that either level can be treated by the housekeeping routines in the same way as a DSA. Chainback fields to the calling block's save areas are held in the head of library workspace and, where more than one level of library workspace is used, a chainback field is set up to the previous level. Figure 3.2 illustrates the method of chaining employed.

ALLOCATION OF LIBRARY WORKSPACE

Library workspace is originally allocated within the program management area by the initialization routine IBMDPII. However, whenever an interrupt occurs and an on-unit is to be entered, a further two levels are allocated. This allows library modules to be called within an on-unit, without overwriting library workspace which may have been in use at the time of interrupt.

Attached to each allocation of library workspace, including the initial allocation in the program management area, is an ON communications area (ONCA). This is a control block used in error handling to hold condition built-in function values. ONCAs are described fully in chapter 7.

Library Modules and Weak External References

Because of the modular structure of the library, a group of modules is frequently used to carry out some particular task. Conversions, for example, are normally done by using a series of modules, and the same is true of many of the mathematical built-in functions. For this reason, many library modules contain a number of external references to modules which may not be needed in a particular program. An example of this is shown in figure 3.3. To prevent unnecessary modules being link-edited, "weak external references" (WXTRNs) are used. WXTRNs are a special type of external reference designed to cater for this situation.

Those entry points that are called only optionally are coded as WXTRNs. This prevents the linkage editor from loading these modules unless a separate external reference is made to them by the compiler. Thus the executable program phase does not contain modules that it never uses.

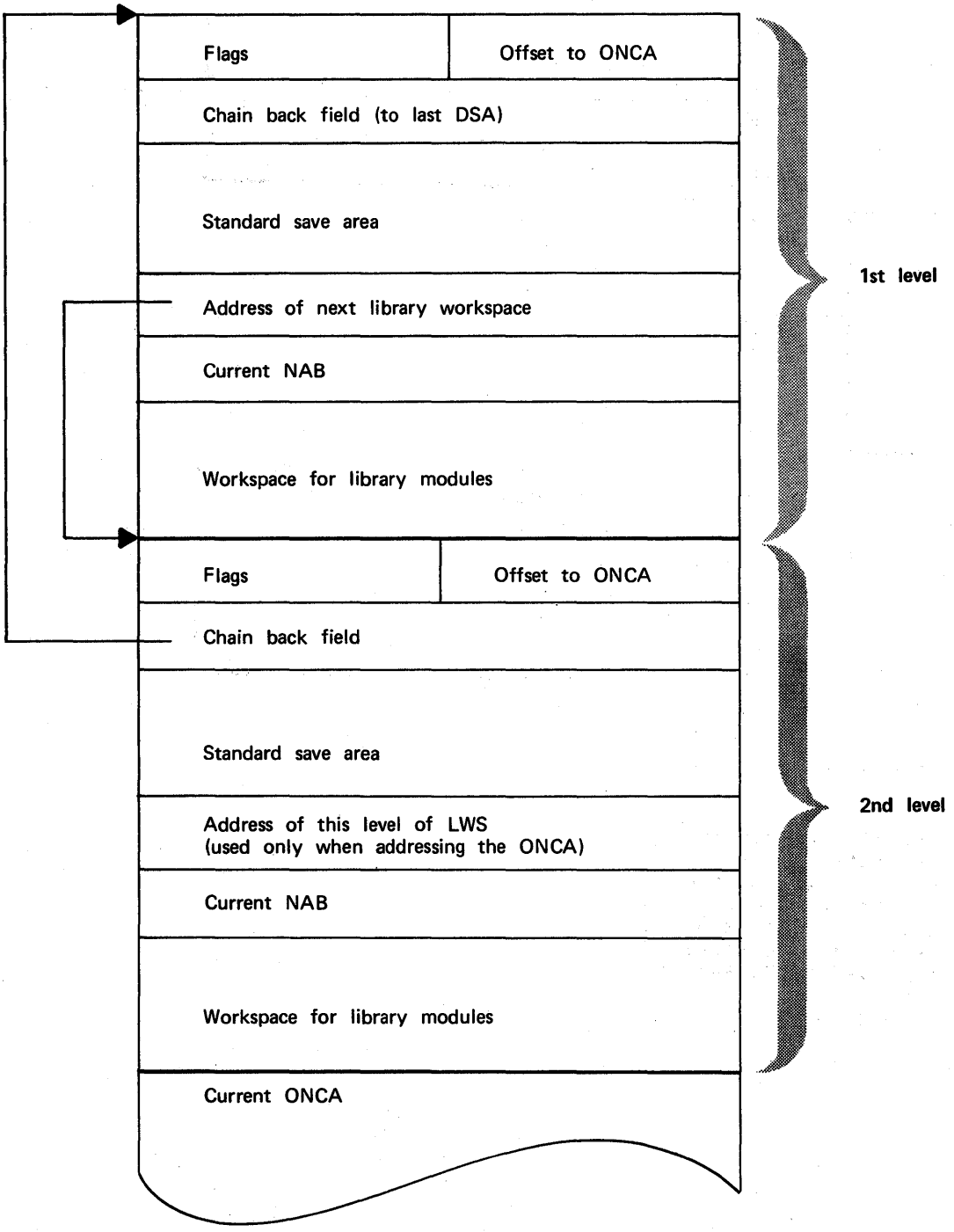


Figure 3.2. Library workspace

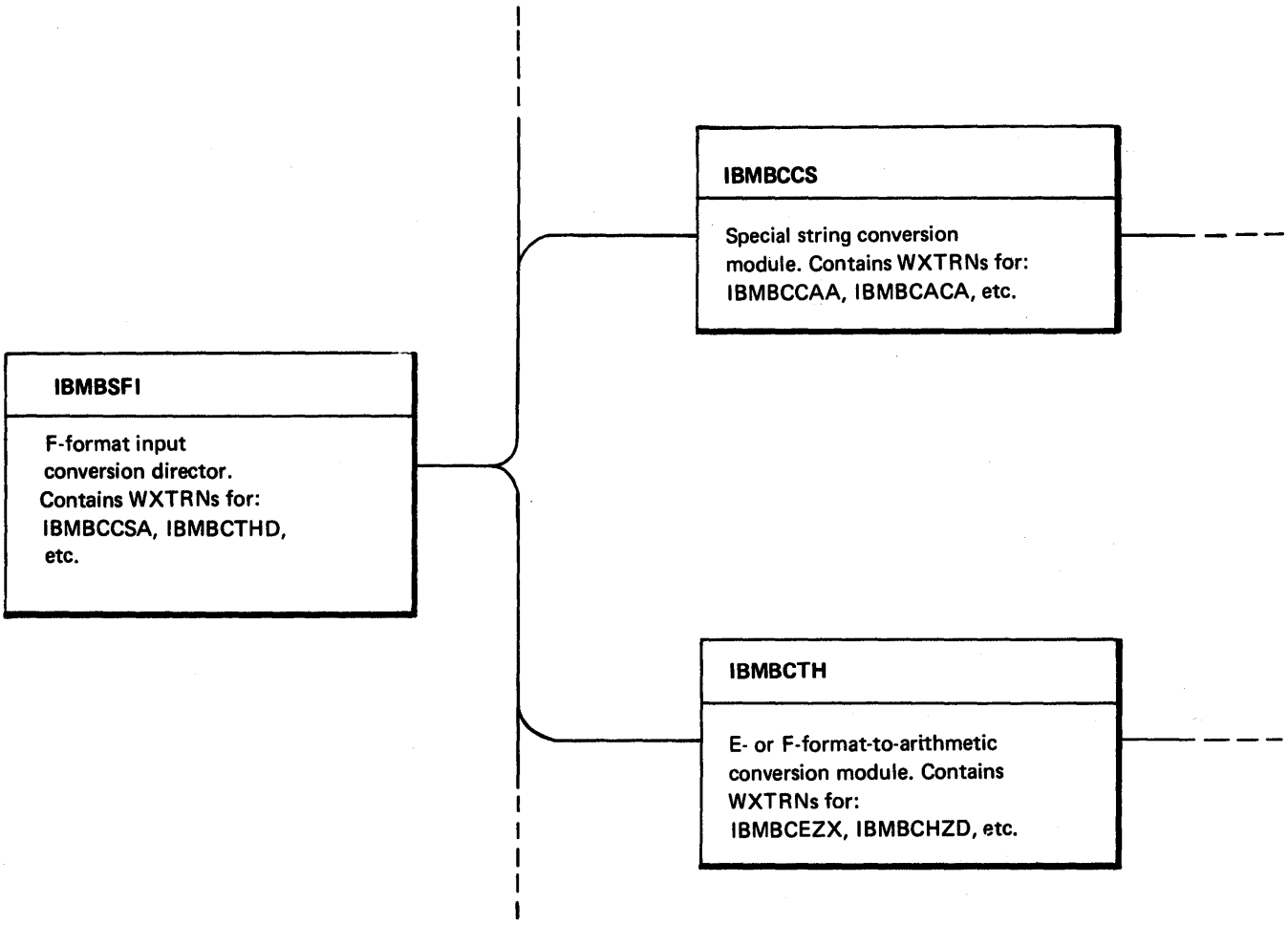


Figure 3.3. Example of use of WXTRNs

Figure 3.3 shows part of a hierarchy of modules with alternative paths through them. When such a hierarchy exists, the actual path to be taken through the modules will be known to the compiler, and external references will be made to all the required modules whose names are coded as WXTRNs. The effect of this is that the linkage editor loads only these modules.

Chapter 4: Communication Between Routines

PL/I allows the programmer the choice of a large number of data attributes. Normally there is no need for explicit attribute information to be retained until execution, because the methods used to handle the data can be resolved during compilation. However, there are certain situations where this cannot be done. For example, the attributes of the data may not be fully known at compile time, because of adjustable bounds or lengths, or the data may be passed to another PL/I program or PL/I library subroutine. When these situations arise, it is necessary to retain some or all of the data attributes in an explicit form throughout execution.

The names of variables fall into a similar category. Normally, they need not be explicitly known during execution. However, for data-directed input/output and the CHECK condition, the names of the variables need to be known so that they can be associated with the correct values.

When such information must be retained until execution, special control blocks are set up for the purpose. These control blocks are described in this chapter. The following control blocks are used.

Descriptors: These hold the extent of the data item (i.e., string lengths, array bounds, and area sizes).

Locators: These hold the address of a data item and are either concatenated with the descriptor, or hold the address of the descriptor.

Descriptor Descriptors: These hold the logical structure levels, dimensions, and lengths, of all elements within a structure.

Data Element Descriptors (DEDs): These hold the attributes of a variable required for data manipulation, except for extents, which are held in descriptors.

Symbol Tables: These hold the names of the variables and associate them with the appropriate storage locations during execution.

Symbol Table Vector: This associates symbol tables with the block in which they are known.

An example of the way in which data is related to its locators, descriptors, and DEDs is given in figure 4.1.

Notes on Terminology

The following terms are used in this chapter.

Virtual origin (VO)	The address where the element of an array whose subscripts are all zero is held or, if such an element does not appear in the array, where it would be held.
Actual origin (AO)	The byte address of the first item in the array or structure.
Relative virtual origin (RVO)	Byte actual origin minus virtual origin.
Structure element	A minor or major structure that contains a number of base elements.
Base element	A data element or array within a structure.

DESCRIPTORS AND LOCATORS

Descriptors are generated when adjustable extents are involved, or when an item is to be passed as an argument and the associated parameter is the type that can be declared with an asterisk among its attributes. For example, DCL X CHAR (N); or DCL X CHAR (*); would both result in the generation of a descriptor. In the first case, code for the SUBSTR built-in function would have to be interpretive if STRINGSIZE were enabled. The appropriate library module would be called, and it would make use of the descriptor to discover the length of the string. This length would have been placed in the descriptor by the prologue code of the block in which the string was declared. In the second case, where the length of the string is signified with an asterisk, the program that is passed the string will expect to receive the length of the string in a descriptor.

Data items that can be declared with an adjustable value or an asterisk are: string lengths, array bounds, and area sizes. Descriptors are, therefore, needed for strings, arrays, and areas. They are

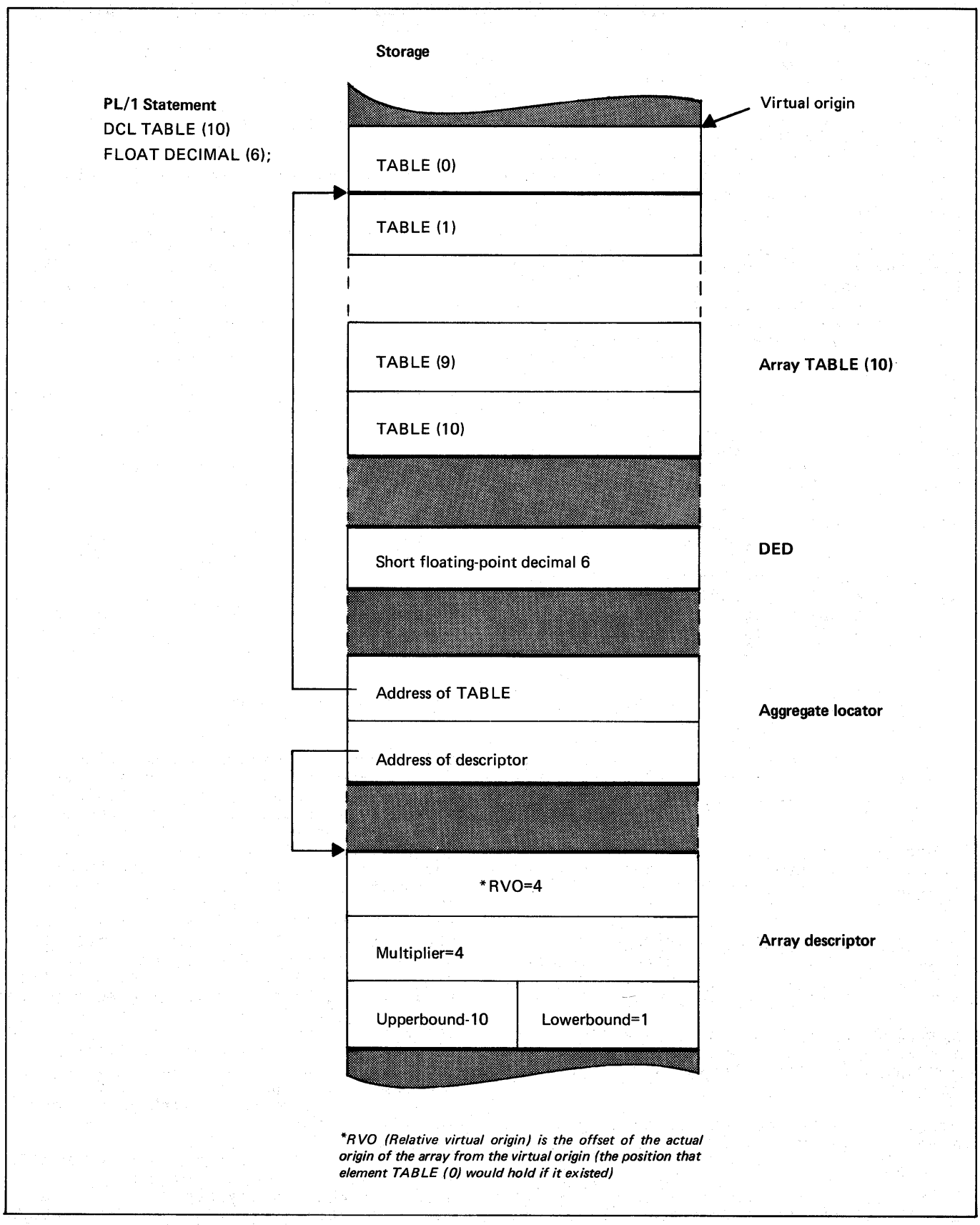


Figure 4.1. Example of descriptors, locators and DEDs for an array

Name of control block	Conditions under which it is generated	Location (control section)
Data element descriptor (DED)	When conversion or stream I/O library modules are called.	Static internal
Array descriptor	When an array has adjustable bounds or may be passed to a library subroutine or other PL/I routine.	Static internal
Aggregate locator	When structure or array descriptor is generated.	Static internal
Area locator/descriptor	When an area is declared with an adjustable size or may be passed as an argument.	Static internal
String locator/descriptor	When a string is declared with an adjustable length or is passed as an argument.	Static internal
Structure descriptor	When a structure is declared with adjustable elements or is passed as an argument.	Static internal
Aggregate descriptor descriptor	When a structure contains elements declared with adjustable bounds.	Static internal
Symbol table	When an item may appear in data-directed I/O or in a CHECK list	Static internal for internal items. Separate CSECT for external items.
Symbol table vector	When GET DATA or PUT DATA is used without a data list, or when SIGNAL CHECK is used without a data list.	Static internal

Figure 4.2. Descriptors, locators, and symbol tables: when generated, where held

also needed for structures, because structures can contain strings, arrays or areas.

In order to connect the data with its descriptor, a further control block is generated. This is the locator. The locator addresses both the descriptor and the variable. For strings and areas, the locator is concatenated with the descriptor and contains only the address of the variable. For structures and arrays, the locator is a separate control block and holds the address of both the variable and the descriptor. Called routines are normally passed the addresses of locators, rather than the addresses of arguments when arguments requiring descriptors are passed.

When the descriptor and locator are not concatenated, it is possible to use the same descriptor for a number of different data items, provided that these items have the same attributes. This process is known as "commoning" and is used to conserve space. Where possible, the compiler commons structure and array descriptors and aggregate descriptor descriptors.

Descriptors and locators are always held in the static internal control section, regardless of the attributes of the data that they describe.

The following types of descriptor and locator are generated. Figure 4.2 summarizes the conditions under which they are generated and gives their storage locations. In the main, they are set up during compilation and completed during execution, if necessary.

String Locator/Descriptor

The string locator/descriptor holds the byte address of the string, information on whether or not it is a varying string, and

the maximum length of the string. For a bit string, the bit offset from the byte address is held. (See figure 4.3.)

Area Locator/Descriptor

The area locator/descriptor holds the address of the start of the area and the length of the area. (See figure 4.4.)

Aggregate Locator

The aggregate locator holds the address of the start of the array or structure and the address of the array descriptor or structure descriptor. (See figure 4.5.)

Array Descriptor

The array descriptor holds:

1. The relative virtual origin (RVO) of the array. This is the offset of the start of the first element in an array (actual origin) from the virtual origin. The virtual origin (VO) is the point at which element (0) would be held in a one-dimensional array, element (0,0) would be held in a two-dimensional array, etc. In a one-dimensional array, the address of any particular element can be discovered by multiplying together the subscript and the multiplier (see below) and adding the result to the virtual origin of the array. An extension of this method is used for multi-dimensional arrays, the formula being:

   Address of element $(S_1, S_2, \ldots, S_n)$

   $= VO + (M * S)$

   where S is the subscript number, and M the multiplier, of the ith dimension, and VO is the virtual origin.

2. The high and low bounds for the subscripts in each dimension.

3. The multiplier for each dimension. The multiplier is the distance between the start of one element and the start of the next element in the same dimension. For example in the array declared A(2,2), the multiplier for the first dimension is the distance between the start of element A(1,1) and the start of element A(1,2).

When the array is an array of strings or areas the string or area descriptor is concatenated with the end of the array descriptor to provide the necessary additional information. Array descriptors are commoned where possible. That is, one descriptor is used for a number of similar arrays. (See figure 4.6.)

Structure Descriptor

This consists of a series of fullwords, giving the byte offset of the start of each base element from the start of the structure. If a base element has a descriptor, the descriptor is included in the structure descriptor, following the appropriate fullword offset. Where a bit offset is involved, this will be held in the descriptor for the bit string, or in the relative virtual origin if the item is a bit string array.

A structure must be mapped during execution if any of the elements in the structure have adjustable bounds or extents, or if the REFER option is used. Where possible, structure descriptors are commoned. That is, one descriptor is used for a number of similar structures. If a structure or an array of structures contains elements with adjustable extents, the structure descriptor is not set up during compilation. Instead, it is set up during execution from information held in the aggregate descriptor descriptor. (See below for information on arrays of structures and structures of arrays.)

Aggregate Descriptor Descriptor

When a structure cannot be mapped during compilation, more information than is held in the structure descriptor is needed for it to be mapped during execution. This information is held in a control block known as an aggregate descriptor descriptor.

The information held in an aggregate descriptor descriptor is the dimensionality and logical level of all the structure

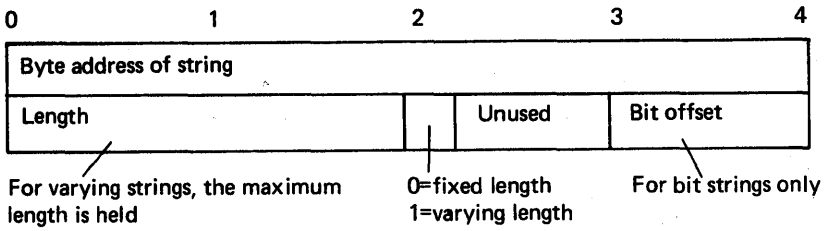


Figure 4.3. String locator/descriptor (SLD)

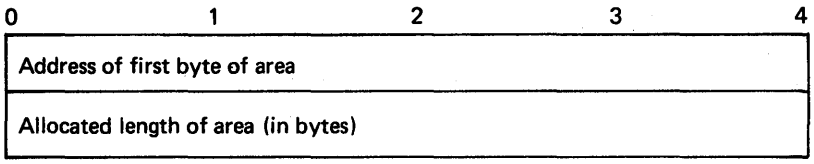


Figure 4.4. Area locator/descriptor (ALD)

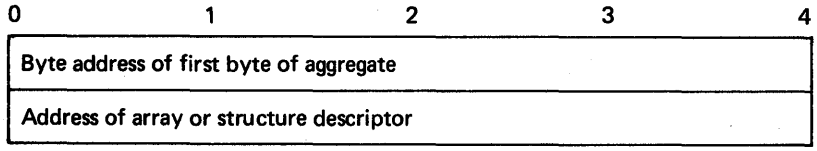
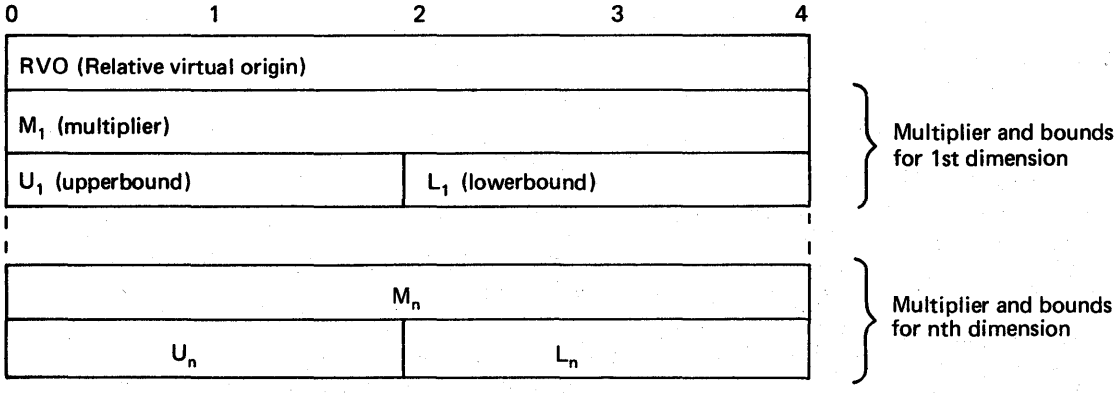


Figure 4.5. Aggregate locator (AL)



Notes:
1. For unaligned bit strings, RVO and multiplier are bit values.
2. For strings and areas, the area or string descriptor is concatenated to the end of the array descriptor.

Figure 4.6. Array descriptor (AD)

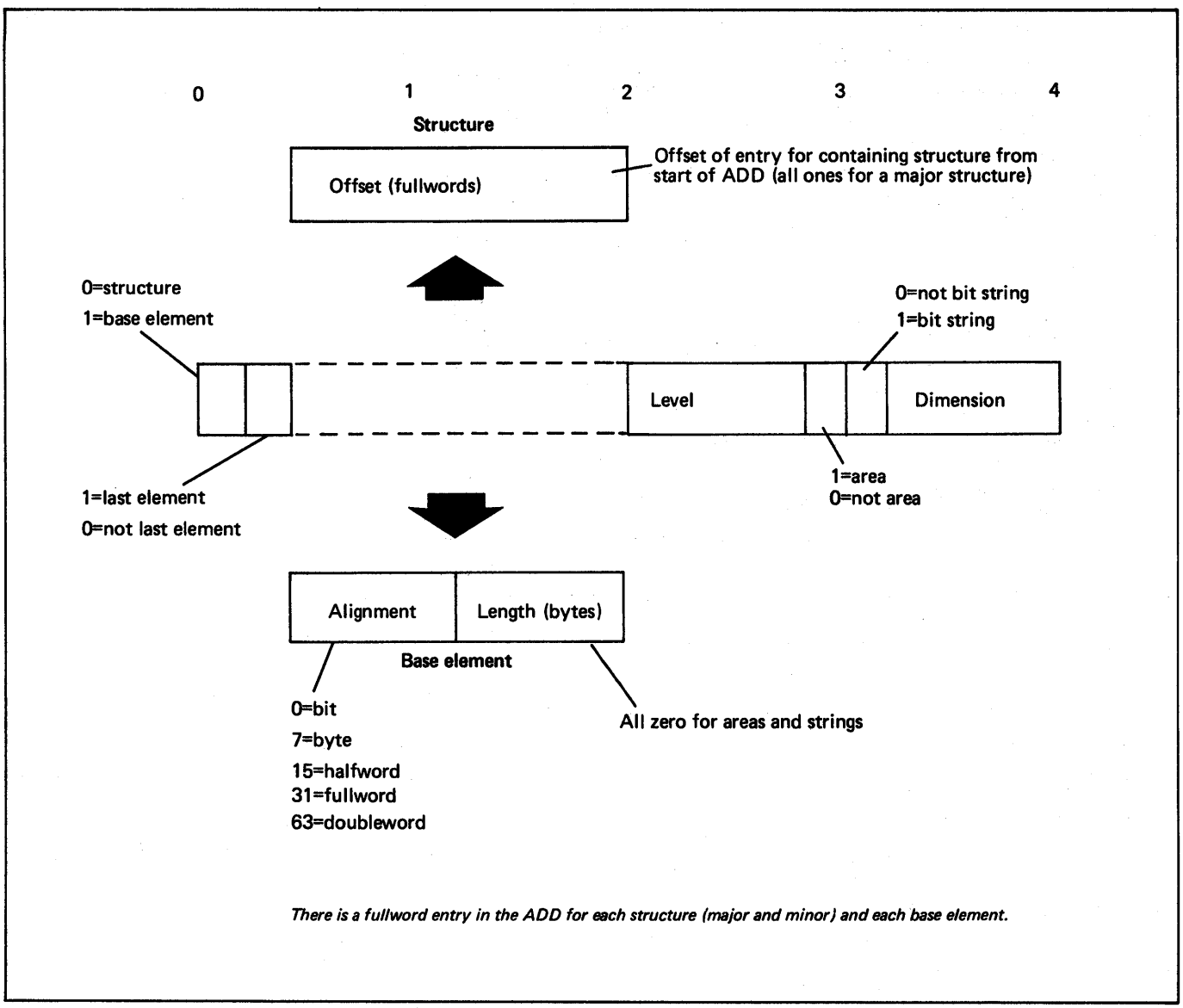


Figure 4.7. Aggregate descriptor descriptor (ADD)

elements, and the dimensionality, logical level, and alignment requirements, of all base elements, plus the length of those base elements that do not have their length held in descriptors. (Strings and areas, and arrays of strings and areas, have their lengths in descriptors.) The length held for an array is the length of an array element. The total length of the array can be calculated by using the information in the array descriptor.

The aggregate descriptor descriptor is set up in static internal storage and is set up completely during compilation. The format is shown in figure 4.7. An example showing the method used to map a structure that contains an element with an adjustable extent is shown in figure 4.8.

Where possible, aggregate descriptor descriptors are commoned.

Arrays of Structures and Structures of Arrays

Where necessary, an aggregate locator, a structure descriptor, and an aggregate descriptor descriptor are generated for both arrays of structures and structures of arrays.

The structure descriptor for both an array of structures and a structure of arrays has the same format. The difference is in the values in the fields of the array

DURING COMPILATION

1. Space for structure descriptor allocated in static storage.
2. Aggregate descriptor descriptor allocated, and fields filled in from structure declaration.
3. Aggregate locator allocated, and address of structure descriptor place in second word.

   Code is generated within the prologue of the block in which the structure is declared to call structure mapping routine, IBMBAMM, to acquire a VDA, and to complete the aggregate locator.

DURING EXECUTION

4. Prologue code places value of N(1 byte) in the string descriptor for D in structure descriptor.
5. IBMBAMM is called to map the structure, using the information in the ADD and the SD (which contains the length of element D). D is aligned with E, then B is aligned with DE. (The rules for structure mapping are given in the language reference manual for this compiler.) The results of the mapping are placed in the structure descriptor.
6. IBMBAMM returns the length of the structure to compiled code, which acquires a VDA for the structure and places the address of the structure in the aggregate locator.

THE RESULT

Every member of the structure can be addressed by means of the address in the aggregate locator and the offsets within the structure descriptor. When bit offsets are involved, they are contained within the appropriate descriptor in the structure descriptor.

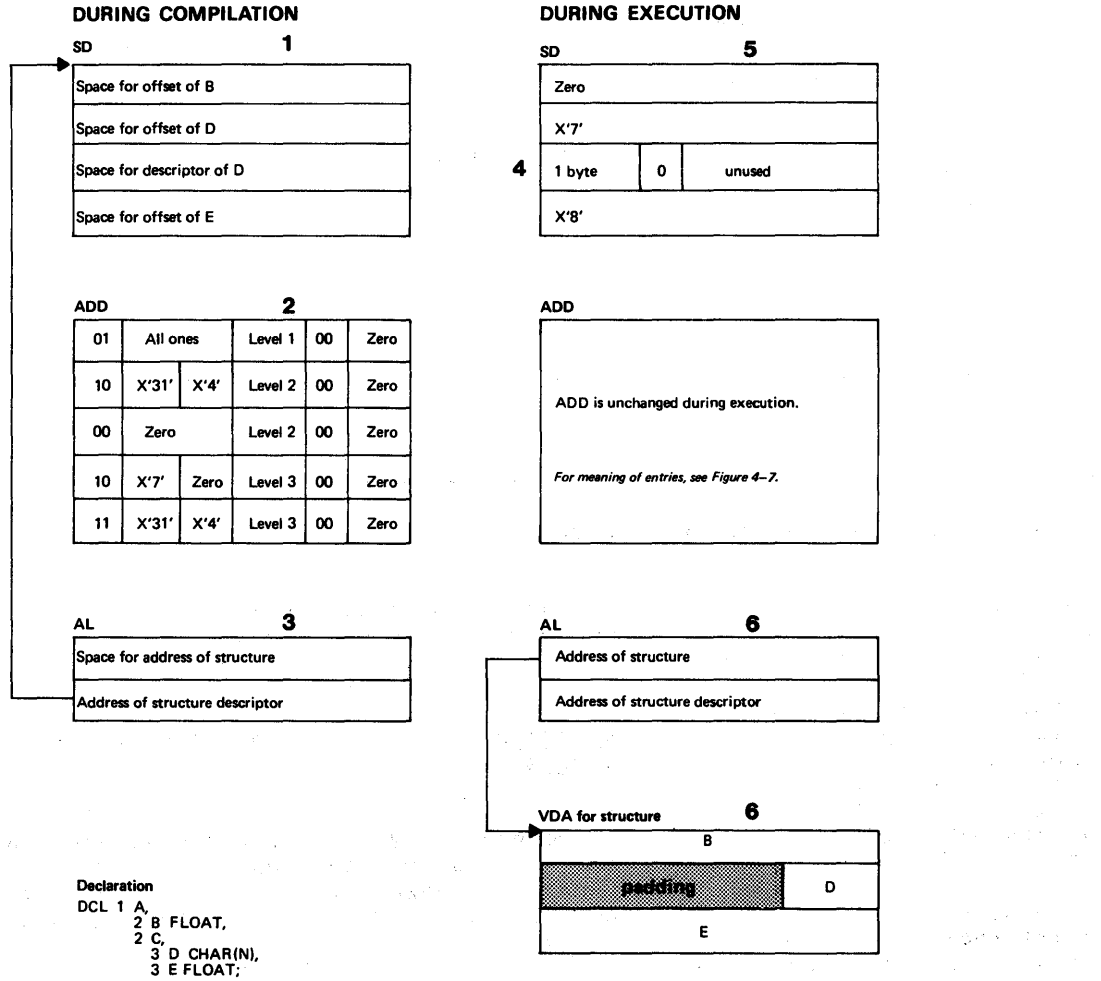


Figure 4.8. Example of handling structure containing adjustable extent

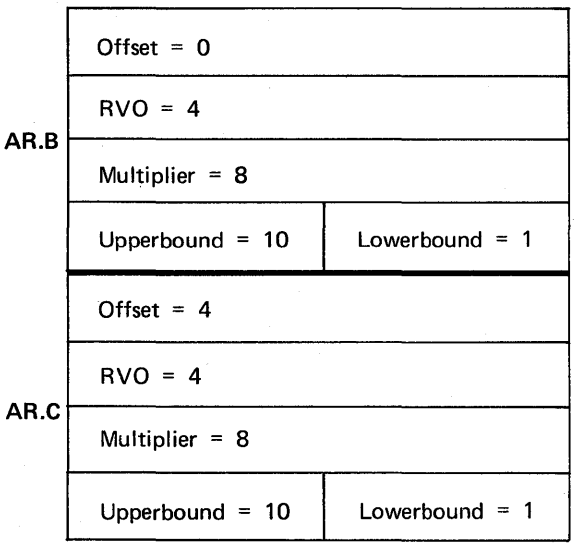
Array of structures

```
DCL 1 AR(10),
      2 B,
      2 C;
```

Structure of Arrays

```
DCL 1 ST,
      2 B(10),
      2 C(10);
```

Structure descriptor for AR



Structure descriptor for ST

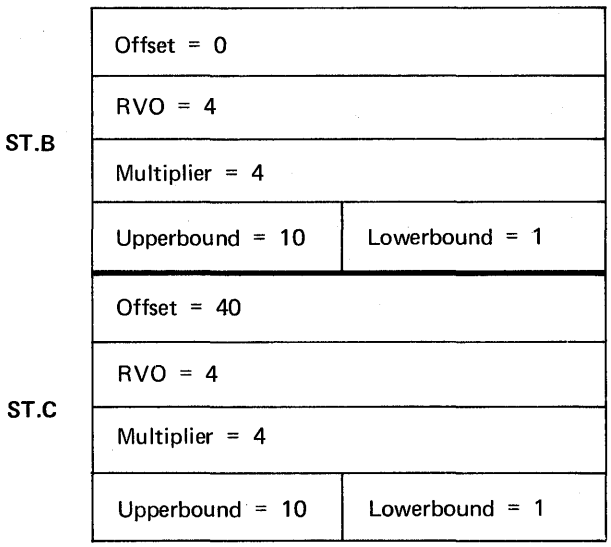


Figure 4.9. Structure descriptors for arrays of structures and structures of arrays

descriptors within the structure descriptor. Take for example the array of structures AR and the structure of arrays ST, declared below.

Array of Structures	Structure of Arrays
DCL 1 AR(10), 2 B, 2 C;	DCL 1 ST, 2 B(10), 2 C(10);

The structure descriptor for both AR and ST would contain an offset field for both B and C and an array descriptor for both B and C. (See figure 4.9.) However, the values in the descriptors would differ, because the array of structures AR would consist of elements held in the order B,C,B,C, etc., and the elements in the structure of arrays ST would be held in the order B,B,B,B,B,B,B,B,B,B,C,C,C,C,C,C,C,C,C,C.

DATA ELEMENT DESCRIPTORS

When data is passed to the PL/I library routines, a complete description of the data is frequently required, and something more than a descriptor is therefore needed. Conversion routines, for example, need to know the complete attributes of the data. To hold such information, data element descriptors (DEDs) are generated. (Control blocks known as DEDs are also used by the compiler. These are compile-time DEDs and have a different format from those that are used during execution. Compile-time DEDs never appear in the executable program.) For stream I/O, DEDs are generated to describe the format of the input or output. These DEDs are known as format element descriptors (FEDs).

DEDs are produced for all types of variable or temporary that are passed to the library for conversion or stream input/output. The length and format of the DED is dictated by the data type of the item. DEDs are shown in detail in appendix B.

DEDs are always held in static internal storage. They are used only to pass information to library routines.

There are five types of DEDs: arithmetic DEDs, arithmetic pictured DEDs, string DEDs, pictured string DEDs, and FEDs.

Arithmetic DEDs: are 4 bytes long.

Arithmetic pictured DEDs: (always decimal) are 8 bytes plus picture specification, which consists of at least one byte for every character in the pictured string. Maximum length for pictured arithmetic DEDs is 264 bytes.

String DEDs: are 4 bytes long.

Pictured string DEDs: (always character string) are six bytes plus the picture specification, which consists of one byte for every character in the picture string. The maximum length for pictured character DEDs is 261 bytes.

FEDs (input/output DEDs): fall into five classes

1. A,B, and control format FEDs have four bytes.
2. E and F format FEDs are six bytes long.
3. Pictured arithmetic FEDs consist of four bytes followed by the pictured arithmetic DED.
4. Pictured character string FEDs consist of four bytes followed by the pictured character string DED.
5. C format FEDs are four bytes plus the two constituent FEDs that make up the complex item. They are used for complex data.

The first two bytes of any DED are the look-up byte and the flag byte. Taken together, they define the data type and permit a receiving routine to determine if it needs to look further into the DED for more information. The general format of DEDs is shown in figure 4.10. Full details are given in appendix B.

SYMBOL TABLES AND SYMBOL TABLE VECTORS

Data-directed I/O statements, and the CHECK condition, require the names of variables to be available throughout execution. Normally, such names are not used after compilation. When required during execution, these names are held in control blocks known as symbol tables or, sometimes, symtabs. Symbol tables hold the name of the variable, its address, and the address of its DED plus certain other information (see appendix B).

PUT DATA and GET DATA statements without a data list, and SIGNAL CHECK statements when there is no check list, imply that the names of all variables known at that point in the program must be available. This information is held in a further control block known as the symbol table vector. The symbol table vector holds the addresses of symbol tables arranged in order of program blocks, commencing with the main procedure block. The symbol table vector consists of a series of fullword fields. These fields contain either the address of a symbol table, a fullword of zeros, or a further address within the symbol table vector. The end of entries for symbol tables, for variables declared in each block, is followed by a fullword of zeros, which in turn is followed by the address in the symbol table vector where entries for the encompassing block begin. If there is no encompassing block, another word of zeros is used.

Figure 4.11 shows the relationship between variables, symbol tables, and the symbol table vector.

Data-directed I/O modules, and the CHECK module, use symbol tables and symbol table vectors in the following ways.

PUT DATA (A,B,C), GET DATA (A,B,C), SIGNAL CHECK (A,B,C): In all these cases, the addresses of the symbol tables for A, B, and C are passed to the appropriate library module.

GET DATA, PUT DATA, SIGNAL CHECK: When no data or check list is included in the statement, the library is passed the address of the start of the associated block entries for the symbol table vector. By following the symbol table vector, it is possible to access the names of all the variables known in the block.

The contents of symbol tables vary according to the storage class of the variable. The method used for holding the address, and other information, is given in appendix B. For internal variables, symbol tables are held in static internal storage. For external variables, symbol tables are held as separate control sections in static external storage. The name of each control section is the name of the associated variable followed by an *. Thus the control section for the external variable B would be B*. Such a control section would also contain the DED of the variable (or DEDs if the variable was a structure).

String DED

Look-up byte	Flag byte	Not used

Arithmetic DED

Look-up byte	Flag byte	Precision	Scale

Pictured string DED

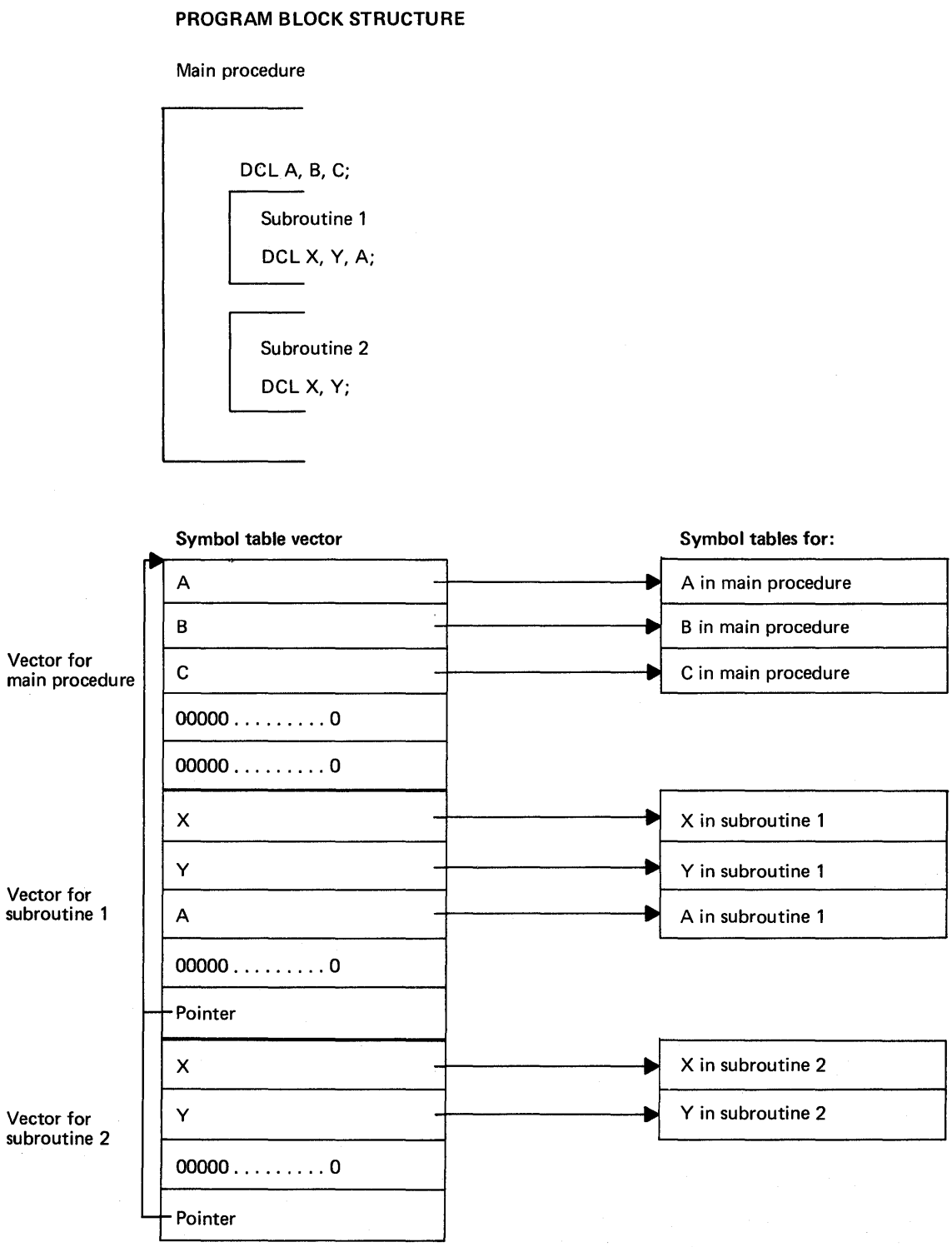
Look-up byte	Flag byte	Length of string
Length of string without/insertion characters		Translation of picture
specification into internal format (one byte per character)		

Pictured arithmetic DED

Look-up byte	Flag byte	Precision	Scale
Length of picture	Length of data	Mantissa byte	Exponent byte
Translation of picture specification into internal format (at least one byte per character)			

For details of look-up byte and flag byte conventions, see appendix B.

Figure 4.10. Format of DEDs



The symbol table vector is built up on a block by block basis, the last entry for each block being a word of zeros followed by a pointer to the first entry for the encompassing block. This mechanism allows for multiple declarations of names.

Figure 4.11. Symbol tables and symbol table vectors

Chapter 5: Object Program Initialization

Before the output from the compiler can be executed as an executable program phase, it must be link-edited, and the PL/I environment must be set up. This chapter briefly describes the effects of link-editing, the manner in which the program is entered, and the initialization process that sets up the PL/I environment.

Link-Editing

The logic and effects of the linkage editor program are described in the publication IBM System/360 DOS: Introduction to System Control Programs. This chapter describes the effects of link-editing on the PL/I program. The linkage editor combines the various control sections generated by the compiler and resolves addresses within these control sections. The linkage editor also incorporates into the executable program phase all library modules that are called from compiled code, and a number of other library modules that are required either because they in turn are called by the library modules called by compiled code, or because they are needed for program management. The most important of the modules used in program management are the error-handling module, IBMDERR, and the storage management module, IBMDPGR. An external reference to both of these modules is contained in the PL/I initialization routine, IBMDPIR. An external reference to IBMDPIR is included in the control section PLISTART which is generated by every compilation and nominated as its entry point. PLISTART contains an external reference to the control section PLIMAIN (which holds the address of the start of the main procedure).

One of the features of the linkage editor is that it does not accept more than one control section with the same name; the second use of the name is ignored. As a result of this, only one PLISTART and one PLIMAIN is generated for each executable program phase. This allows two or more PL/I main procedures to be link-edited together. The procedure that receives control will be the first that is passed to the linkage editor, because it will be the PLISTART and PLIMAIN of this procedure that are included in the executable program phase. This feature is also used to handle data declared EXTERNAL. Control sections for each such data item are generated by all programs in which the data is declared. Only one of these is resolved.

The PLIMAIN control section is not generated by the compiler if the PL/I source program does not contain the MAIN option. Instead, a control section named PLIMAIN is included in the initialization module IBMDPIR. This control section contains the address of code that calls the module IBMDPEP, which puts out a message saying there is no main procedure and then terminates the program.

Program Initialization

Code is compiled by the PL/I Optimizing Compiler on the assumption that various control blocks will be set up and that certain registers will point to them when the program is entered. This arrangement of control blocks and registers is known as the PL/I environment.

The most important factors affecting the PL/I environment are the following:

1. A dynamic storage area (DSA) should exist before compiled code is entered. This will give the address of the area available for the first compiled code DSA and will act as a save area for the calling routine's registers.

2. A task communications area (TCA) should exist. The TCA acts as a central communications area for the program, holding addresses of various storage- and error-handling routines, and control blocks. The TCA also contains a number of flags and other fields.

3. Program checks should be passed to the PL/I error-handling module IBMDERR.

4. Pre-formatted DSAs should exist for certain library routines. These pre-formatted DSAs are known as library workspace (LWS).

5. A space should be available for any condition built-in function values (ONCHAR, ONSOURCE, etc.) should a PL/I interrupt occur. This space is known as an ON communications area (ONCA). As the condition built-in functions have default values, an area to hold the default values is required. This is known as the dummy

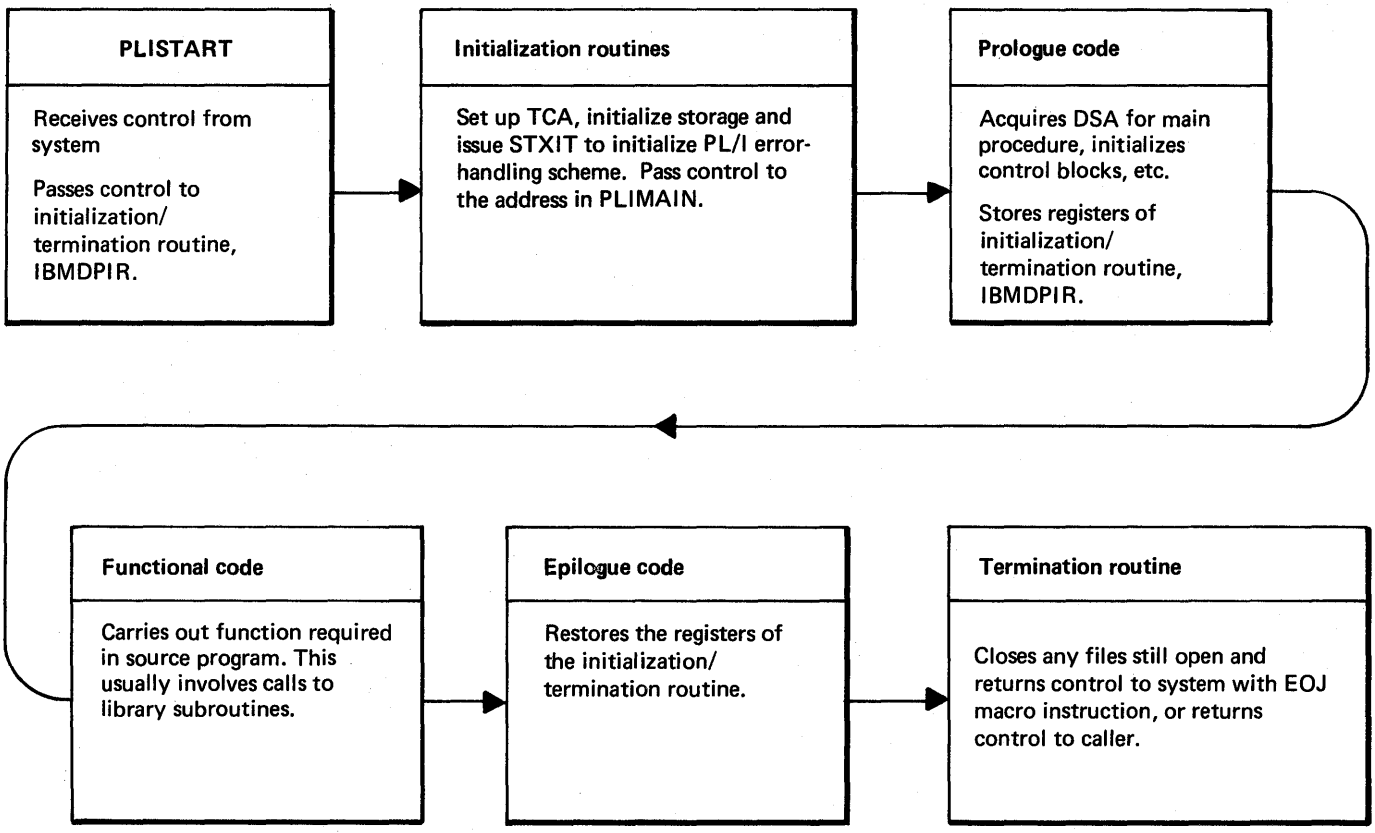


Figure 5.1. Flow of control during execution

ONCA.

6. Register 12 should point at the TCA, and register 13 should point to the DSA.

The resident program initialization routine IBMDPIR, and the transient routine IBMDPII, which it calls, set up the various control blocks involved immediately following the executable program phase in an area known as the program management area. The contents of the program management area are described later in this chapter.

Two similar initialization procedures are available for initializing the PL/I environment when a PL/I procedure is called from assembler language. These modules are IBMDPJR which is resident and carries out the same functions as IBMDPIR, and IBMDPJI which is transient and carries out the same functions as IBMDPII. The use of these modules prevents the normal PL/I program having an overhead of redundant code.

The advantage of having program initialization routines is that it obviates the need for special code in the prologue of main procedures and allows two procedures with the MAIN option to be used in one program. As shown in figure 5.1, the initialization routine IBMDPIR is re-entered after the execution of compiled code. Again, this is done automatically by standard epilogue code. This is because the registers of IBMDPIR have been stored by the prologue code and are restored by the epilogue code. The functions of IBMDPIR and IBMDPII are explained below.

INITIALIZATION AND TERMINATION ROUTINES

When called from the control program, IBMDPIR established the initial storage area (ISA) in that part of the partition that is not taken up by the executable program phase. It then calls the transient routine IBMDPII which sets up the program management area and issues a STXIT macro instruction so that program checks will be passed to the error handler. Control is

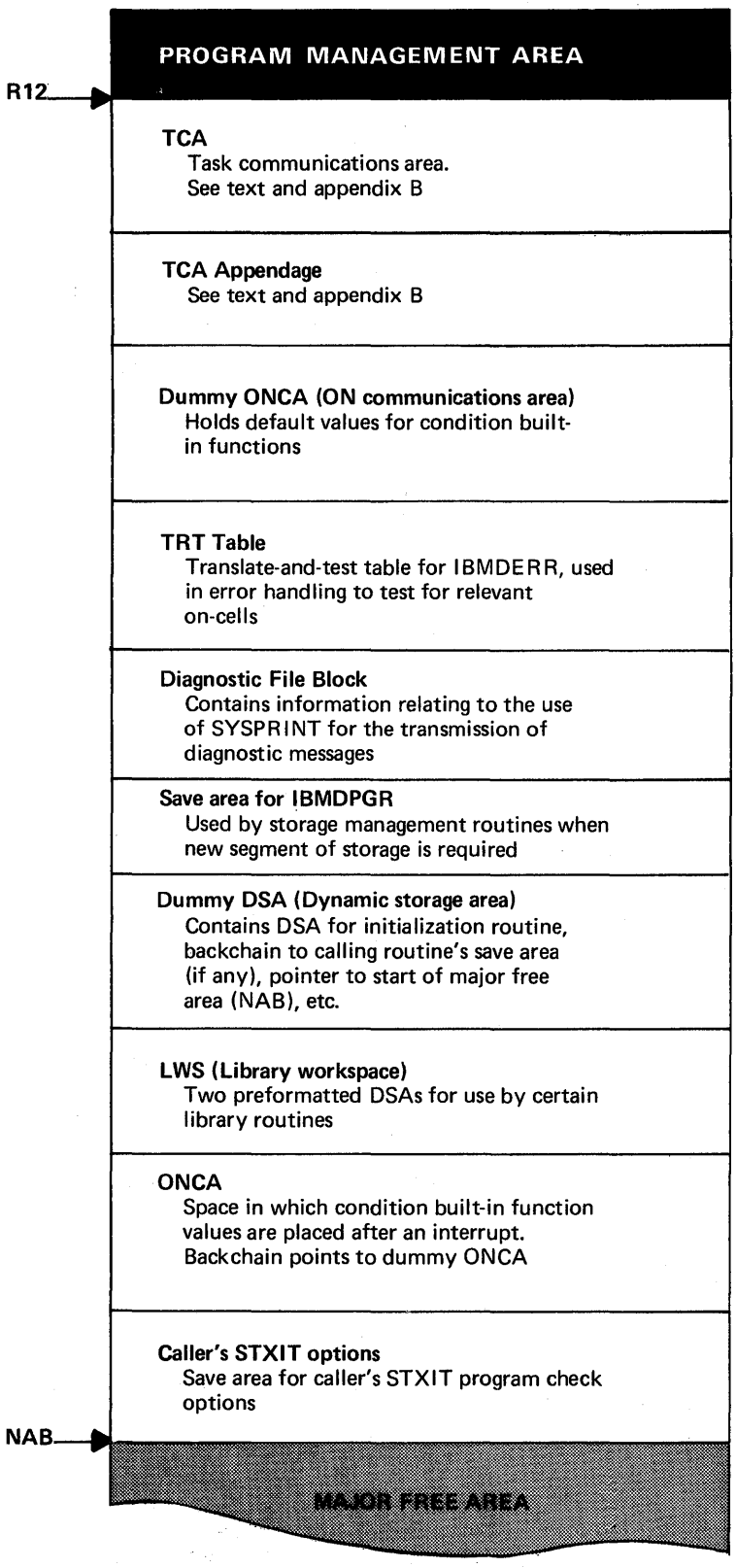


Figure 5.2. Program management area

then returned to IBMDPIR which passes control to the procedure whose address is held in the control section PLIMAIN. Before passing control, register 13 is pointed to the dummy DSA and register 12 to the TCA. The dummy DSA will be used by compiled code to store IBMDPIR's registers.

On return from compiled code, IBMDPIR raises the FINISH condition by calling the error handler IBMDERR. After handling the FINISH condition, IBMDERR branches to the GOTO code in the TCA to terminate the program. This is standard system action for the FINISH condition and for normal return from an ON FINISH on-unit. The GOTO code is given the address of the dummy DSA as the target DSA address. The abnormal GOTO out of block routine is thus enetered. This routine checks to see if any files are open and closes any that are. Any exit DSA processing (for example termination of SORT) is handled during this routine. Finally a test is made to discover whether IBMDPIR was called from the control program. If so, a test is made to discover whether the termination is the result of an ERROR condition. If the termination is caused by an ERROR condition, a CANCEL macro instruction is issued to ensure the flushing of any SYSIPT data. If the termination is not caused by an error condition, an EOJ macro instruction is issued to return to the control program. If there was a caller, IBMDPJR is called at entry point IBMBPJRC (see below).

IBMDPIR also contains certain utility routines that may be used by all programs. These are the GOTO out of block routine described in chapter 2, and the code to acquire second and subsequent levels of library workspace. IBMDPIR also contains a CSECT PLIMAIN that is link-edited if an attempt is made to initialize a non-main PL/I procedure. This dummy PLIMAIN is link-edited because the compiler only generates a PLIMAIN control section for a main procedure. The dummy PLIMAIN contains code to load and call the module IBMDPEP. IBMDPEP puts out a message indicating that there is no main procedure and terminates the program.

Initialization when a PL/I procedure is called from another language is similar except that the module IBMDPJR is used. This module contains the entry points PLICALLA and PLICALLB.

The major difference between IBMDPIR and IBMDPJR is that IBMDPJR can be passed parameters indicating the size and the location provided for the ISA, and also a parameter list for the called procedure.

IBMDPJR has four initialization entry points as follows:

PLICALLA — ISA acquired is remainder of partition. Can pass parameter list to PL/I program. Control passed to procedure whose address is in PLIMAIN.

PLICALLB — ISA size and address must be specified. Can pass parameter list to compiled code. Control passed to procedure whose address is in PLIMAIN.

IBMBPJRA — ISA acquired is remainder of partition. Can pass parameter list to compield code. Control passed to routine whose address is in word addressed by register zero.

IBMBPJRB — ISA size and address must be specified. Can pass parameter list to compiled code. Control passed to routine whose address is in word addressed by register zero.

A further entry point IBMBPJRC is used during termination. If the GOTO cut of block routine discovers that PL/I was called from another language it returns control to IBMBJRC which calculates a return code and restores the caller's STXIT options and program mask. IBMBPJRC then returns to the caller.

IBMDPJR calls on the transient routine IBMDPJI to initialize the program management area, save the caller;s STXIT options and set up the PL/I STXIT options.

When calling PL/I from another language via PLICALLA or PLICALLB it is necessary to explicitly include reference to IBMDPJR in the input to the linkage editor. For example by an INCLUDE IBMBPJRA statement. (Note that IBMBPJRA must be specified because this is the entry point name.)

THE PROGRAM MANAGEMENT AREA

A diagram of the program management area is shown in figure 5.2. It shows the situation when the compiled program is called. The various fields in the program management area are shown in detail in appendix B. A brief description of their use is given below.

Task Communications Area (TCA)

The TCA is the central communications block used throughout the program. It is used to address the error-handling and storage-management routines, and to point to the current segment of dynamic storage.

The TCA is the most important control block in the PL/I environment. A field-by-field description follows.

Flags — Indicate that an abnormal GOTO cut of block may take place (see below). Also indicate that certain special error conditions may arise.

BOS — The pointer that points to the beginning of the current segment (see chapter 6).

EOS — The pointer that points to the end of the current segment (see chapter 6).

Address of external save area:

The address of the save area for the calling routine, if IBMDPIR was not called from the control program.

Address of translate-and-test table for IBMDERR:

See below, under heading "Translate-and-Test Table."

Address of TCA appendage

Address of save area for IBMBPGRC and IBMBPGRD (see below)

Open file chain:

Used when closing files at end of job

PL/I and user return code:

A standard area to keep these codes.

Address of IBMBPGRD:

Stack overflow routine for VDAs (see chapter 6)

Address of the diagnostic file block (see below)

Address of flow statement table:

This is used to address the flow statement table which holds statement numbers for use during execution.

Address of tab table:

The address of a table of

tabulator positions used in list-directed output.

Address of FLOW module:

The address of the module used to implement the compiler FLOW option.

Address of storage-handling routines:

Entry points to IBMDPGR that get non-LIFO storage, free non-LIFO storage, and acquire a new segment for LIFO storage (see chapter 6).

Address of IBMBERRB

Address branched to after a software-detected interrupt occurs (see chapter 7).

Address of environment descriptor:

Identifies release of compiler being used.

Code for GOTO out of block:

Whenever a GOTO out of block occurs, or could potentially occur because of the value of a label variable, compiled code branches to this code in the TCA, which calls a subroutine in IBMDPIR. The subroutine calls the flow and count module if either FLOW or COUNT is in effect and tests flags to see if an abnormal GOTO has occurred. The abnormal GOTO flags are set by compiled code or by library routines in which an abnormal GOTO could occur. (See, for example, description of SORT in chapter 11.) If the abnormal GOTO flag is set, the abnormal GOTO subroutine is called. Otherwise, the routine restores the value of register 13 for the block to which a branch is being made, and also restores the values of registers 4 (the temporary base) and 3 (the static base). Condition enablement is also restored, and NAB may require alteration if a variable data area (VDA) has been used. If necessary IBMBPGO is called to reset CHECK enablement. Finally, a branch is made to the correct location and the program base altered if this is necessary.

Address of count module

Used to call IBMBEFLC when the COUNT option is in effect.

Address of IBMBPGO

Used in GOTO code see above

Addresses of various routines:

The TCA is completed with the address of the WAIT COMPLETION and Event assign routines. These are library routines that, for speed, are addressed from the TCA. The addresses are held as WXTRNs and are resolved if compiled code calls these modules.

The fields to hold the address of the overflow routine, the environment descriptor, the GOTO code, and the error handler are duplicated. This allows version 4 of the compiler to use more efficient means of calling these routines and retains compatibility with previous versions. For details of the locations of TCA fields see Appendix B.

TCA Appendage (TIA)

The TCA appendage is addressed from the standard part of the TCA (see above). Its contents are as follows:

Address of the byte beyond the ISA (TISA):

This holds the address beyond the end of the partition and is necessary because EOS gets altered when non-LIFO dynamic storage is allocated.

Address of last free area (TLFE):

This points to a chain of areas of non-LIFO dynamic storage that have been freed, but cannot be amalgamated with the major free area.

Flags Indicating:

1. SYSPRINT is opened for

stream output (i.e., that it has been opened as expected and can be used for error messages).

2. That an abnormal termination is in progress.

3. That dump I/O is in progress.

Address of dummy DSA:

Used, when abnormally terminating the program, to restore IBMDPIR's registers. This is faster than chaining back and testing for the dummy DSA, and allows IBMDPIR to be reached should the DSA chain become overwritten.

Address of get-library-workspace routine:

This routine is part of the transient library module IBMDPIR and is used to get a new allocation of library workspace and an ONCA. This routine is called after interrupts and during program initialization (see chapter 3).

Address of loaded-module chain:

This is a chain of the names of transient library modules that have been loaded. It is kept to prevent duplicate loading.

Code to handle interrupts and save area for STXIT:

The STXIT macro requires a save area to use after a program check interrupt (program check); this is the area used. The code branches to the error-handling module IBMDERR.

Address of interrupt handler (TERA):

This is the address to which the branch is made after a program check interrupt (see above) has occurred.

Address of count tables

These are used for implementing the COUNT option.

Address of the TCA

This is used to restore register 12 when a program check occurs, in case it has been changed by a non-PL/I subroutine.

Tab Table

Forty bytes reserved for PLITABS (transient library module IBMBSTAB). This CSECT is loaded when a STREAM PRINT file is opened or when PLIDUMP is called. It contains the linesize, tabulating positions and other information for PRINT files. Only the pagesize is used for PLIDUMP.

Save Area for IBMDPGR

This area is used as a DSA for IBMDPGR, the routine entered when there is not enough room for a further DSA in the current segment of the LIFO stack. Both DSAs in library workspace may be in use when IBMDPGR is required, and there may be no caller's save area because a DSA has not yet been acquired. Consequently, IBMDPGR has a save area reserved in the program management area.

Dummy ONCA

The dummy ONCA holds default values for the condition built-in functions. These will be supplied if they are requested either when no interrupt has occurred, or when no interrupt with the requested condition built-in function value has occurred. There is a chain back through all ONCAs to the dummy ONCA. (See chapter 7.)

Translate-and-Test Table

The translate-and-test table contains code used in error handling to identify relevant on-cells. (See chapter 7.)

Diagnostic File Block

The diagnostic file block holds information used by the error-message modules. This includes the address of the SYSPRINT transmitter.

Dummy DSA

The dummy DSA acts as a save area for the registers of the initialization routine IBMDPIR, and an end to the chain of DSAs when a search through blocks is being made, as for example, when searching for a relevant established on-unit (see chapter 7). The dummy DSA has a bit in its flag byte to indicate that it is a dummy. The dummy DSA contains a NAB (next available byte) pointer enabling the main procedure to obtain a DSA in the LIFO stack.

Library Workspace (LWS)

This consists of two pre-formatted DSAs that are used by certain of the library modules. (See chapter 3.)

ON Communications Area (ONCA)

The ONCA is supplied as an area where compiled code or library routines can store or read out any condition built-function values that may be required. (See chapter 7.)

Caller's STXIT Options

This is two words in which the caller's STXIT PC options are held. These are restored before a return is made.

Operation Interrupt Analysis Code

This is code placed in the program management area by IBMDPII. It is used by the error handler to test whether an operation interrupt has been caused by an attempt to execute a floating point instruction on a machine with no floating point hardware.

During program initialization IBMDPII tests to see if the machine has floating point hardware. If it has, the code consists only of a direct return. If there is no floating point hardware on the machine, code to analyze the interrupt is placed in the program management area.

The code is addressed from the TCA and is called by the error handler when an operation interrupt occurs. If the analysis shows that the error was due to lack of floating point hardware, a code is returned to the error handler.

Chapter 6: Storage Management

The program compiled by the DOS PL/I Optimizing Compiler is executed in either a background partition or a foreground partition. The executable program phase is placed at the start of the partition. The remainder of the partition is known as the initial storage area (ISA), and is used for various functions during execution. The start of the ISA is used as the program management area. This is an area that contains a number of housekeeping fields and is set up by the initialization routine IBMDPIR. (See chapter 5.) The remainder of the ISA is used for PL/I dynamic storage allocation.

The contents of the executable program phase are described in chapters 2 and 5 of this publication. The contents of the program management area are described in chapter 5. This chapter is concerned with the allocation and freeing of PL/I dynamic storage. Information on storage handling during interlanguage communication is given in chapter 13.

TYPES OF DYNAMIC STORAGE REQUIRED

The requirement for dynamic allocation and freeing of storage is inherent in the language. Automatic variables are allocated and freed on a block-by-block basis. Controlled and based variables can be allocated and freed by appropriate PL/I statements. Storage is also obtained dynamically for workspace, compiler-generated temporary values, I/O buffers, and PL/I transient library routines.

Dynamic storage can be conveniently divided into two classes.

1. That which is allocated and freed on a last-in/first-out (LIFO) basis.
2. That which is not.

The first class is known as LIFO dynamic storage and the second class as non-LIFO dynamic storage.

Contents of LIFO Storage

Two kinds of storage area are allocated in LIFO storage. They are dynamic storage areas(DSAs) and variable data areas(VDAs).

A DSA is allocated for every procedure or block and contains:

- The operating system standard save area.
- Certain standard housekeeping fields.
- All automatic variables and compiler-generated temporaries whose length is known during compilation.

VDAs are acquired for all other allocations of LIFO dynamic storage. These are:

- Storage for automatic variables and compiler-generated temporaries whose length is not known until execution. (DCL X CHAR(N), for example.)
- Storage for those transiently-loaded library routines that can be freed immediately they have been used. (Open and dump routines, for example.)
- Library workspace (LWS) and ON communication area (ONCA) acquired immediately before an on-unit is entered.
- Workspace for certain library modules.

Contents of Non-LIFO Storage

Non-LIFO storage is used for the following:

- Controlled variables.
- Those based variables that are allocated by the ALLOCATE statement, provided that they are not allocated in a static or automatic area.
- Such transient library routines as may be required more than once when they have been loaded. (I/O transmitters, for example).
- Input/output buffers.

Dynamic Storage Allocation

The principle used in dynamic storage allocation is to allocate LIFO storage from the low-address end of ISA, starting at the first 8-byte boundary beyond the program management area, and to allocate non-LIFO

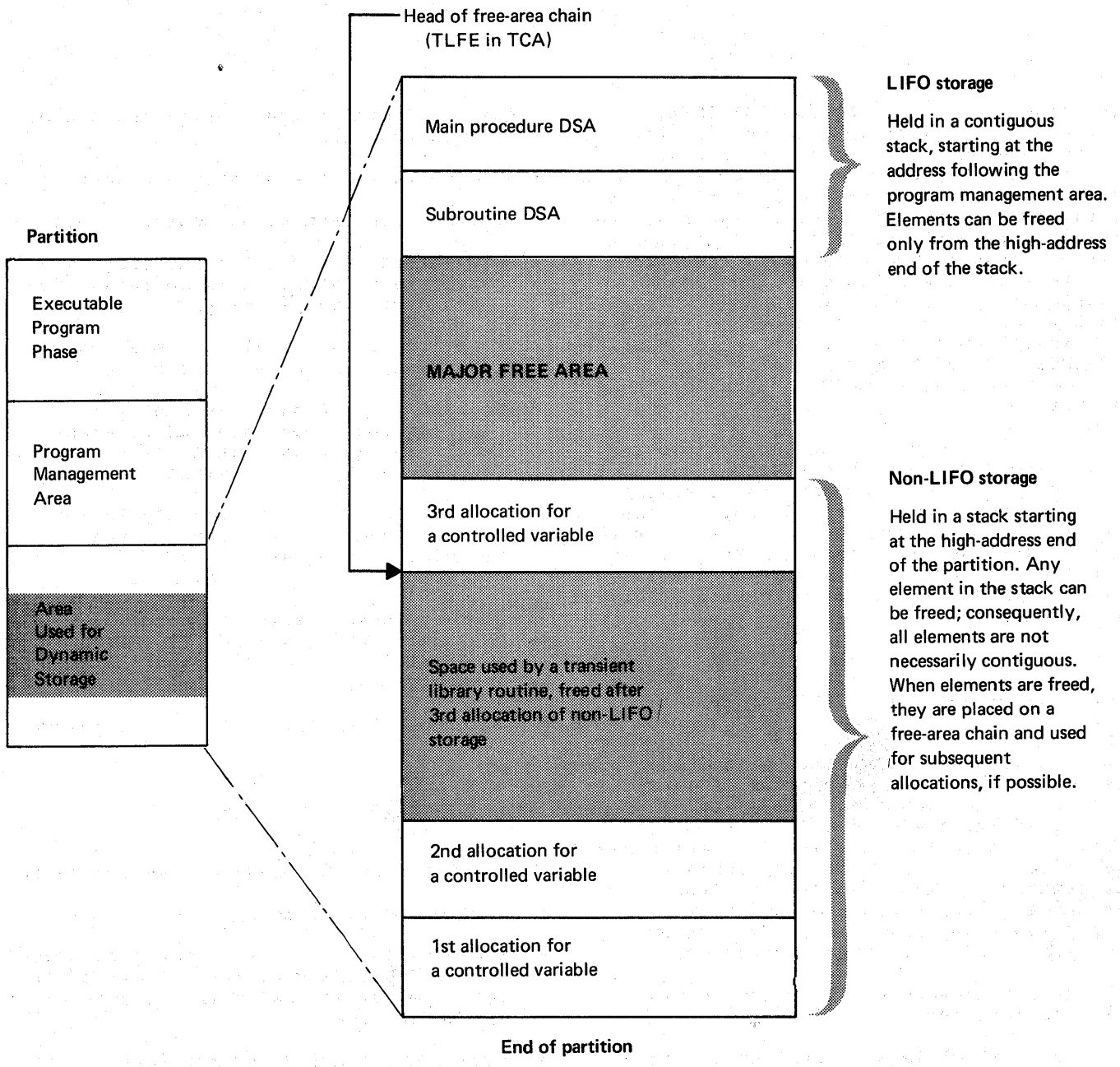


Figure 6.1. The principles of dynamic storage allocation

storage from the high-address end of the ISA, which is also the high-address end of the partition. All storage is allocated in multiples of 8 bytes. Between the areas of LIFO and non-LIFO storage is an unused section known as the major free area. (See figure 6.1.)

The last element in the LIFO stack is always the first to be freed and consequently can always be amalgamated with the major free area. This is not always the case with non-LIFO storage. When an item not contiguous with the major free area in the non-LIFO stack is freed, it is placed on a free-area chain whose head is anchored in the TCA. Attempts are always made to use areas on this chain when further allocations of non-LIFO storage are made.

Allocations of LIFO storage are made by testing to see if there is enough space in the major free area. If there is not

enough space, an attempt is made to use an area on the free-area chain. When an area on the free-area chain is used, it is known as a new segment of the LIFO stack.

Fields used in Storage Handling

To keep track of the storage allocated and freed, a number of pointers are used. These are:

- The begining-of-segment pointer (BOS).
- The end-of-segment pointer (EOS).
- The next-available-byte pointer (NAB).
- The free-area chain.
- A pointer to the byte beyond the end of the ISA (TISA).

The beginning-of-segment pointer (BOS) is initially set during program initialization to point to the start of the ISA. It is not altered unless a new segment of storage is acquired. BOS always points to the start of the current storage segment. BOS is held at offset 8 from the head of the TCA, and is addressed from register 12.

The end-of-segment pointer (EOS) is initially set during program initialization to point to the end of the ISA. However, it is updated, when non-LIFO storage is allocated, to point to the end of the major free area. EOS is held at offset X'C' (12) in the TCA, and is addressed from register 12.

The next-available-byte pointer (NAB) is held in every DSA and points to the first 8-byte boundary beyond the DSA (or VDA if one has been acquired). This address is the start of the major free area. The current NAB is held in the most recent DSA and is addressed from offset X'4C' (76) from register 13. As register 13 is altered every time a DSA is acquired, the value in a NAB pointer need only be altered when a VDA is freed or acquired. Previous NABs are automatically restored when register 13 is pointed to a previous DSA.

The pointer to the byte beyond the ISA (TISA) is used to keep track of the end of the ISA.

The first two bytes of BOS, EOS, and NAB contain segment numbers ("FF" for the ISA). The use of these numbers is explained under "Acquiring a New Segment."

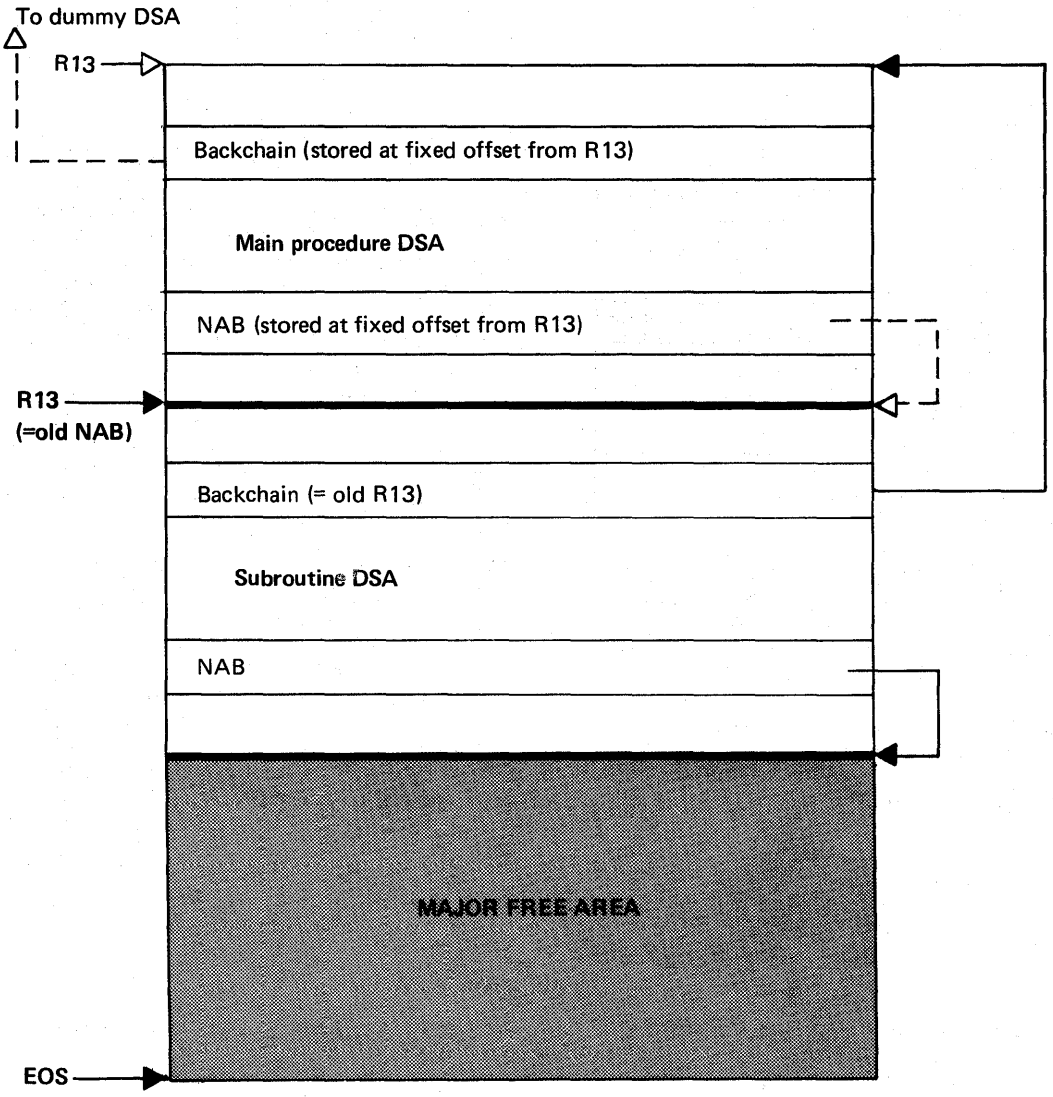
The free-area chain includes those elements of non-LIFO dynamic storage that have been freed but that could not be amalgamated with the major free area. The start of the chain is held at offset 8 (TLFE) in the TCA appendage and points to that element with the highest address.

ALLOCATING AND FREEING LIFO STORAGE

Allocating and freeing LIFO storage is handled by compiled code or by the particular library module that requires the space. The allocation is done in the manner used by the prologue code shown in chapter 2. Freeing is done in the manner used by the epilogue code, which is also shown in chapter 2. Before allocating the storage, a test is made to see if there is enough space in the major free area for the new allocation. This test is carried out by logical arithmetic, for reasons explained later, under the heading "Acquiring a New Segment." If there is not enough room, entry to the segment-handling entry point of the resident library module IBMDPGR is made. The address of this entry point is held at offset X'74' (116) in the TCA. The allocation of LIFO storage involves finding the current NAB. This gives the address of the start of storage to be used. A new value of NAB is calculated, addressing the byte beyond the end of the new allocation. Freeing the storage is done by restoring the NAB pointer to the previous value. Figure 6.2 illustrates the principles involved. Before allocating the storage, a test is made to see if there is enough space for the allocation.

ALLOCATING AND FREEING NON-LIFO STORAGE

Any section of non-LIFO storage can be freed at any time; therefore a simple stacking mechanism cannot be used, because it would waste storage by leaving freed storage within the stack. A modified method is therefore used. When storage that is contiguous with the major free area is freed, it is amalgamated with the major free area by altering the end-of-segment pointer, which indicates the end of this area. When storage that is not contiguous with the major free area is freed, it is placed on the free-area chain, which is anchored to a field in the TCA. Whenever an allocation is made, an attempt is made to place the allocation in an area that is already on the chain, rather than use a further section of the major free area. Allocations of non-LIFO dynamic storage are always handled by the library module IBMDPGR, whose address is held in the TCA. Figure 6.3 illustrates the principles



Allocating a new DSA

1. Test if major free area large enough for new DSA. If not call IBMBPGRC.
2. Store R13 at fixed offset from old NAB to act as backchain.
3. Load R13 with address of old NAB.
4. Store new NAB at fixed offset from register R13.

Freeing a DSA

1. Load register 13 with current backchain address. Since the NAB and backchain fields are always addressed from register 13, the previous values are automatically restored.

Figure 6.2. Principles involved in allocating and freeing LIFO storage

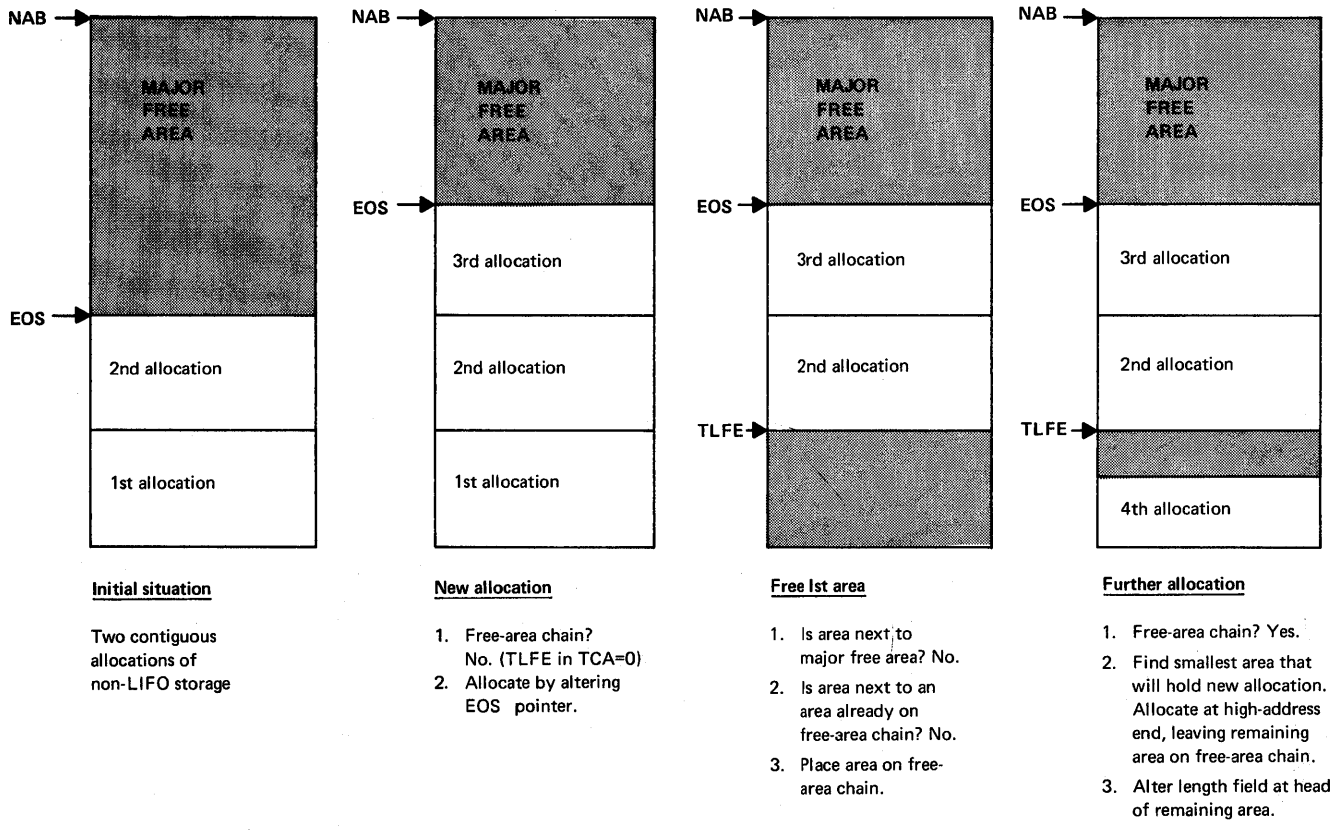


Figure 6.3. Principles involved in allocating and freeing non-LIFO storage

involved. Whenever an allocation within the major free area is made, the end-of-segment pointer, in the TCA, is updated to point to the end of the major free area.

If there is insufficient space for an allocation of non-LIFO storage either in the major free area or in an area on the free-area chain, the program is terminated.

ACQUIRING A NEW SEGMENT OF LIFO STORAGE

Every time a new procedure or block is entered, or a VDA is acquired, a test is made to see whether there is enough space, for the DSA or VDA, between the NAB pointer and the EOS pointer. If there is <u>not</u> enough space then an attempt is made to use the largest space on the free-area chain.

Pointers BOS and EOS are set to point to the beginning and end, respectively, of the new segment. The DSA or VDA is allocated in the low-address end of the segment, and the NAB pointer is set to point to the first free byte after the DSA or VDA. The former values of BOS and EOS are stored at the start of the new segment.

A segment number is given to each segment, starting at hexadecimal "FF" and reducing by 1 for each new segment. The number for the ISA is "FF", the second segment "FE", and so on. This number is held as the first byte of the NAB, BOS, and EOS pointers. The result of this device is that when logical arithmetic is used, all addresses in later segments are apparently less than those in the earlier segments, regardless of their actual position. This simplifies segment handling. For instance, when a DSA in the second segment is freed,

NAB is simply restored to its previous value which may well be in the first segment. NAB will then hold value "FF------", and EOS the value "FE------". When a further DSA is required, EOS will be less than the sum of NAB and the DSA length, as NAB is already greater than EOS. Consequently it will appear that there is insufficient space for the DSA in the first segment, regardless of whether or not this is the case. The library module IBMDPGR is thus called to restore BOS and EOS, rearrange the free-area chain, and if possible place the new DSA after NAB in the first segment. The process is illustrated in figure 6.4.

IBMDPGR - STORAGE MANAGEMENT ROUTINE

The allocation and freeing of LIFO storage within a given segment is handled by compiled code or by the library module requiring the storage. All other dynamic storage allocation is carried out by the resident library routine IBMDPGR; this module has four entry points:

IBMBPGRA	Allocating non-LIFO storage.
IBMBPGRB	Freeing non-LIFO storage.
IBMBPGRC	Obtaining and freeing additional storage segments (for DSAs).
IBMBPGRD	Obtaining and freeing additional storage segments (for VDAs).

These four entry points are described below.

Allocating Non-LIFO Storage (IBMBPGRA)

When entered by entry point IBMBPGRA, the module first frees any LIFO segments that are not in use, searches the free-area chain, if one exists, and allocates the storage in the smallest possible area on the chain. If an area of the exact size is found, it is removed from the chain; otherwise the length stored in the first word of the area is altered. If there is no chain, or no area on the chain that is large enough, IBMDPGR attempts to allocate the storage in the area immediately preceding the EOS pointer. If there is not enough space between the EOS pointer and the current NAB pointer, the program is terminated.

Provided that storage can be allocated, control is passed back, with register 1 pointing to the address of the storage allocated.

Freeing Non-LIFO Storage (IBMBPGRB)

When entered by IBMBPGRB, the module scans the free-area chain (if one exists) to see whether the storage being freed can be amalgamated with areas already on the chain. This is done if possible. The module then checks to see whether the storage being freed is adjacent to the major free area. If so, EOS is altered to point to the end of the area being freed or to the end of the amalgamated area, if this adjoins the major free area. If neither case applies, the area is added to the free-area chain, which is arranged in descending order of addresses.

Segment Handling (IBMBPGRC and IBMBPGRD)

When compiled code discovers that the address contained in the NAB pointer plus the length of the area to be allocated is greater than the value of the pointer EOS, either IBMDPGRC or IBMDPGRD is called, depending on whether a new DSA or VDA is required.

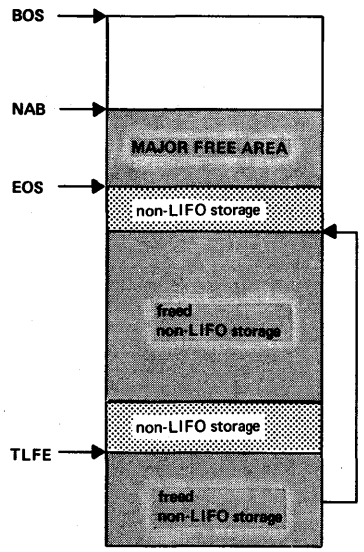
The entry points are called in two circumstances:

1. There is insufficient room in the current segment for allocation of the DSA or VDA and, consequently, a new segment is required.

2. A segment other than the first one has been allocated, but is no longer in use.

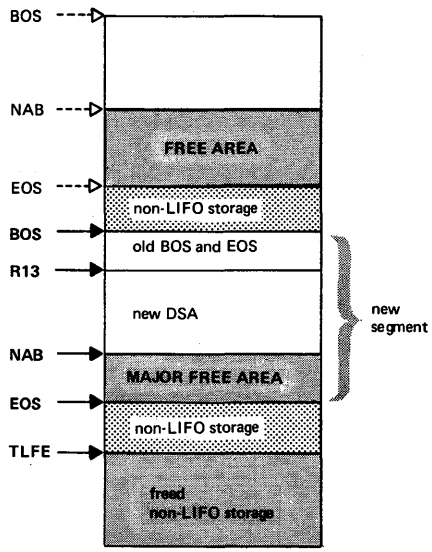
IBMBPGRC and IBMBPGRD check to see which of the above two situations caused the call. This is done by checking whether the number in the first byte of NAB is greater than the number in the first byte of EOS.

In case 1 above, the segment numbers are the same, and a new segment must be allocated. A new segment is allocated by searching the free-area chain for the largest available area and using this as a new segment. If there is no area large enough to hold the new DSA, control returns to IBMDPIR, which calls module IBMDPES to generate an "insufficient storage" message, and then terminates the program. If a new segment is allocated, the old values of BOS and EOS are placed in control words at the head of the new segment. New values for BOS and EOS, with first byte numbers



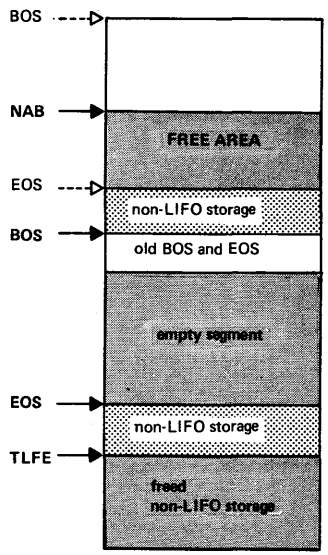
Initial situation

1. Free-area chain exists. BOS, NAB, and EOS have X'FF' in first byte, i.e., segment number 1.



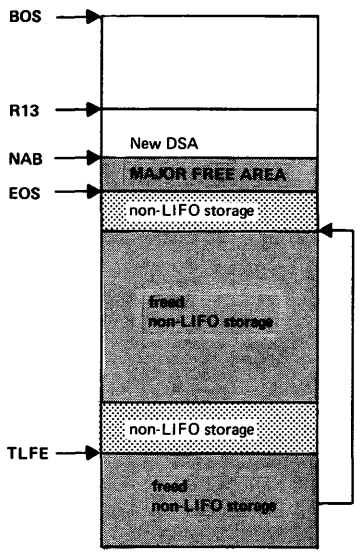
Acquiring new segment

1. Compiled code or library routine finds major free areas too small. Calls IBMBGR.
2. IBMBGR finds an area on free-area chain large enough for allocation.
3. Stores old BOS and EOS. Sets new BOS and EOS, and returns to caller.
4. Caller gets new DSA. BOS, EOS, and NAB have X'FE' in first byte, i.e., segment number 2.



Freeing DSA in segment

1. Register 13 is restored in the normal way. BOS and EOS are not restored. The segment will not be freed until there is a further demand for storage, that can be accommodated in the previous segment.
2. NAB now has X'FF' in first byte, BOS and EOS still have X'FE'.



Freeing segment

1. When storage is again required, NAB + storage required is compared with EOS using logical arithmetic.
2. NAB + storage is found to be greater (because of different segment numbers), so IBMBGR is called.
3. IBMBGR finds segment numbers are different. Tests to see if new storage will fit in old segment. If not, allocates in current segment.
4. Storage will fit, so restores old BOS and EOS, places segment on free-area chain, and returns to caller.
5. Caller allocates storage starting at current NAB.

Figure 6.4. Segment handling

decremented by one, are placed in the TCA. The address of the new NAB is passed in register zero; the address for the start of the new DSA or VDA is passed in register 1.

In case 2 above, the number in the first byte of NAB is greater than the number in the first byte of EOS. If the difference is greater than one, then more than one extra segment has been allocated for DSAs which are no longer current. In this case, segments are freed until only one empty segment remains. This is done by setting BOS and EOS to the values held in the control words at the head of each segment and freeing the storage in the way described for IBMBPGRB above.

When only one empty segment remains, a test is made to see whether the new DSA will fit into the segment that contains the present NAB pointer (the segment before the empty segment). This test is made by comparing the current NAB pointer with the old EOS pointer held in the control words of the empty segment. If there is sufficient room, the empty segment is freed as described under IBMBPGRB, above. Return is then made to the caller with a new value for EOS and BOS, and the DSA is allocated immediately after the old NAB.

If there is not enough room in the segment containing NAB, then a test is made to see if the empty segment is large enough to hold the new DSA. This is done by comparing the difference between the current BOS and EOS with the length of the element. If there is enough room, the DSA is allocated in the empty segment. The address of the start of the storage is passed to compiled code in general register 1 and the address of the new NAB in general register 0.

If there is not enough room in the empty segment, then the segment is freed. There are now no empty segments, and the situation is treated as if there had been no empty segments in the first place.

Note: It is possible that after freeing a number of empty segments, an area on the free-area chain can immediately follow EOS. However, the possibility is remote, and no check is made to see whether this is the case.

Storage Management in Programmer-Allocated Areas

By using area variables, PL/I allows the programmer to use a continuous area of storage for based variables. The allocation of storage for area variables is handled in the same way as that for other types of variable, and depends on the variable's storage class. The allocation and freeing of storage within an area is handled by the library module IBMBPAM.

If there is not enough space for an allocation or if the target area is too small in an assignment statement, the AREA condition is raised.

The method employed is that storage is allocated from the low-address end of the area, and an offset is kept to the end of the item with the highest address allocated in that area. This offset is known as OEE (offset to end of extent). When storage is freed, either the OEE is altered or the storage is placed on a free-storage chain, with the largest segment at the start of the chain. The method used is conceptually the same as that used for non-LIFO storage. However, the chain is held in a different order and the stack extended in the opposite direction. If a space freed is adjacent to any that are already free, the spaces involved are amalgamated into one. This is done either by altering OEE, or combining the new free space with one or more free spaces that are already in the free chain, or possibly by a combination of these methods. When a free chain exists, IBMBPAM always attempts to allocate storage by using a space on the chain. The low-address end of the smallest possible space on the chain is used, and the chain is then rearranged to maintain the correct order of decreasing size.

Chapter 7: Error and Condition Handling

This chapter deals with the method used to implement execution-time error handling. Many errors detected at execution time are related to PL/I conditions and can be handled either by on-units written by the programmer, or by standard system action. The chapter starts with a summary of the error and PL/I condition handling facilities offered by the PL/I language. The implementation problems these facilities raise, and the methods used to handle them, are then described. An example of error handling is given which allows the principles involved to be followed through in a program. The chapter finishes with a brief description of the error message modules, the modules used to implement the PLIDUMP facility, and other debugging facilities.

Summary of PL/I Error Handling

PL/I allows the programmer to obtain control after the occurrence of any PL/I interrupt. A PL/I interrupt is defined as "the redirection of flow of control of a program, as the result of the raising of an enabled PL/I condition." A PL/I interrupt is not the same as a program check interrupt, although all program check interrupts, other than "significance" and certain input/output interrupts, can, and normally do, cause PL/I interrupts. (See figure 7.1.) PL/I interrupts are always associated with PL/I conditions. They fall into three classes:

1. Hardware interrupts (program checks) which are interpreted as PL/I conditions and treated as PL/I interrupts. See figure 7.1.

2. Special PL/I error conditions that have no equivalent at the program check level. For example: CONVERSION and SUBSCRIPTRANGE.

3. PL/I conditions that are not errors, but occur at unpredictable times during the program; for example, ENDFILE and ENDPAGE.

Throughout this chapter, interrupts and PL/I conditions recognized by the system, are referred to as program check interrupts. Interrupts detected by checking code at the PL/I level are referred to as software-detected interrupts, or software interrupts.

A table showing all PL/I conditions, and the method by which they are detected, is given in figure 7.2.

Hardware interrupt	PL/I condition
Operation	
Privileged operation	
Execute	
Protection	ERROR
Addressing	(after issuing
Specification	a message)
Data	
Fixed point overflow	FIXEDOVERFLOW/SIZE
Fixed point divide	ZERODIVIDE/SIZE
Decimal overflow	FIXEDOVERFLOW/SIZE
Decimal divide	ZERODIVIDE
Exponent overflow	OVERFLOW/SIZE
Exponent underflow	UNDERFLOW
Floating point divide	ZERODIVIDE

Figure 7.1. Hardware interrupts associated with PL/I conditions

The programmer has the choice of defining the action that will be taken when a PL/I interrupt occurs, by writing an on-unit for the relevant condition, or of accepting the default action. This default action is known as standard system action. If control has passed through an ON-statement (that has not been overriden by a REVERT or other statement), the associated on-unit is said to be established.

The programmer also has the choice of whether or not certain of the PL/I conditions shall cause PL/I interrupts when they occur. When a condition causes a PL/I interrupt, it is said to be enabled. When it does not cause a PL/I interrupt, it is said to be disabled. Some conditions are enabled by default; some are disabled by default. Details are shown in figure 7.2. Many of the conditions whose enablement is under programmer control are error conditions that are peculiar to the PL/I language. Because these interrupts are not recognized by the system, special code has to be generated if they are to be recognized. PL/I, therefore, gives the programmer a choice of whether to have this code generated and have more efficient debugging aids at his disposal, or of not having the code generated, and having a faster and shorter program. Normally,

relevant conditions will be enabled during the testing of a program and removed for production runs. Conditions whose enablement is under programmer control can be enabled or disabled for the duration of a statement or a block.

Certain of the program check interrupts can be disabled by the programmer. When this is done, the interrupts still occur at system level. They are intercepted before PL/I interrupts occur, and control is returned to the point of interrupt.

A number of further factors influence the implementation of error handling. These factors are discussed below.

Static and Dynamic Scope

On-units have dynamic scope. That is, each procedure or block inherits the on-units established in the block that calls it, unless such on-units are specifically overridden. Thus, as the sequence in which procedures are called may depend on the input data, it is not always possible at compile time to predict which on-units will be applicable to any given section of the program. A method is therefore needed to determine this during execution.

Condition prefixes have static scope. That is, they are inherited from the encompassing procedure or block in the source program. Thus, the enablement or otherwise of a condition is predictable at compile time.

Levels of Interrupt

An important concept in the understanding of error handling is that of levels of interrupt. With a few minor exceptions, the language allows any statements to be executed in on-units, but stipulates that the environment of the original interrupt must be retained so that a return can be made to the point of interrupt when the on-unit is completed. This necessitates a further allocation of library workspace for use during the on-unit, as the original allocation may be in use when the interrupt occurs and cannot be overwritten. As there may be PL/I interrupts during the execution of on-units, the stacking of levels of library workspace can, theoretically, continue indefinitely.

Condition Built-In Functions

PL/I defines a number of condition built-in functions and pseudovariables that allow the program to inspect the causes of an interrupt and, in certain cases, to alter the fields involved. These built-in function values are placed in a special control block known as the ONCA (ON communications area) before the main error-handling code is entered.

Because any number of levels of interrupt can occur, a new ONCA is provided for every level of interrupt. This is done at the same time as a new allocation of library workspace is made.

The ERROR Condition

The ERROR condition is raised as standard system action for those PL/I conditions that will normally be caused only by errors. (See figure 7.2 for details.) It is also raised by certain program check interrupts (see figure 7.1), and a number of software-detected interrupts that have no directly-associated PL/I condition. A message is generated before the condition is raised. This is beyond the control of the programmer, and the message will be produced regardless of the presence of an ERROR on-unit.

The Implementation of Error Handling

The most important points in the error-handling scheme are the following:

1. During compilation, the compiler generates code to check for the various PL/I conditions that are not related to program check interrupts or, alternatively, code to call appropriate library modules that will check for these conditions when they carry out the required function.

2. During program initialization, the initialization module IBMDPII issues a STXIT macro instruction, which results in the system passing control to the error-handling routine IBMDERR, whenever a program check (hardware-detected interrupt) occurs.

3. During execution. The PL/I resident library module IBMDERR gains control whenever a PL/I condition is recognized, or a program check interrupt associated with a PL/I

Name of condition	Qualified	Description	Recognized by	Default	Programmer Control	ERROR** Condition
Computational						
CONVERSION	no	Attempt to convert invalid character string	Code in relevant library modules	enabled	yes	yes
FIXEDOVERFLOW	no	Overflow of a fixed point value	System	enabled	yes	yes
SIZE	no	Attempt to assign too large a value	Compiler-generated checking code, or program system	disabled	yes	yes
OVERFLOW	no	Overflow of a floating-point value	System	enabled	yes	yes
UNDERFLOW	no	Exponent becomes smaller than permitted minimum	System	enabled	yes	no
ZERODIVIDE	no	Attempt to divide by zero	System	enabled	yes	yes
Input/Output						
ENDFILE	yes	End of file reached	Code in relevant library modules	enabled	no	yes
ENDPAGE	yes	End of a page on a print file reached	Code in relevant library modules	enabled	no	no
TRANSMIT	yes	Transmission error on a file	Code in library modules	enabled	no	yes
UNDEFINEDFILE	yes	Error in opening file	Code in relevant library modules	enabled	no	yes
KEY	yes	Invalid key	Code in relevant library modules	enabled	no	yes
NAME	yes	Unrecognizable data-directed input	Code in relevant library modules	enabled	no	no
RECORD	yes	Incorrect size record	Code in relevant library modules	enabled	no	yes

** The ERROR condition is raised when an error occurs that is not covered by PL/I exceptional conditions - taking the square root of a real negative number, for example. It is also raised as standard system action when handling all types of error conditions. Thus an ERROR on-unit enables the programmer to intercept all error conditions.

Figure 7.2. (Part 1 of 2). PL/I conditions

Name of Condition	Qualified	Description	Recognized by	Default	Programmer control	ERROR** condition
Program Checkout						
SUBSCRIPTRANGE	no	Array subscript outside declared bounds	Compiler-generated checking code	disabled	yes	yes
STRINGSIZE	no	Attempt to assign string to smaller string	Code in relevant library modules	disabled	yes	no
STRINGRANGE	no	Attempt to access beyond limits of string	Code in relevant* library modules	disabled	yes	no
CHECK (variable or label)	yes	Value assigned to identifier or control passed through label	Compiler-generated checking code, or library module	disabled	yes	no
List Processing						
AREA	no	Attempt to allocate beyond end of area	Relevant library modules	enabled	no	yes
System Action						
ERROR	no	Any error condition including those not covered by other conditions**	Relevant library modules	enabled	no	-
FINISH	no	Program about to be terminated	Relevant library modules	enabled	no	-
Programmer Named						
CONDITION (name)	no	Programmer defined condition	Signal statement	enabled (when coded)	no	-

* When STRINGRANGE is enabled, library modules are always called to handle substring operations. These modules have the necessary code for checking for the STRINGRANGE condition.

** The ERROR condition is raised when an error occurs that is not covered by PL/I exceptional conditions - taking the square root of a real negative number, for example. It is also raised as standard system action when handling all types of error conditions. Thus an ERROR on-unit enables the programmer to intercept all error conditions.

Figure 7.2. (Part 2 of 2). PL/I conditions

condition occurs. This module checks to see if the associated PL/I condition is enabled and, if it is not, returns control to the point of interrupt. If the condition is enabled, the module then searches for a relevant, established on-unit, passing control to the first one that is found. If no established on-unit is found, IBMDERR carries out the standard system action, calling any library modules necessary to do this.

INFORMATION REQUIRED AT INTERRUPT

For the error-handling module to carry out the various functions necessary, the following information must be available.

1. The type of PL/I condition that has occurred and the associated on-code.
2. Whether the condition is one that can be enabled or disabled by the programmer and, if it is such a condition, whether it is enabled or disabled.
3. Whether there is an established on-unit that applies to this condition.
4. The values that may be used by any on-unit built-in function that appears in the on-unit.
5. The standard system action for the condition.

THE FIELDS USED IN ERROR HANDLING

The fields used in accessing this information are the following: the error code, enable cells, ONCBs (ON control blocks), ONCAs (ON communications areas), and translate-and-test table in the TCA. The fields are shown in figure 7.3 and an example of error handling given in figure 7.4. Appendix B contains diagrams of these fields.

The Error Code

The error code is either a four-byte or two-byte code. A four-byte code is generated when a PL/I condition is detected by compiled code. A two-byte code is generated when an error has been detected that does not have an immediately associated PL/I condition.

Byte 1 holds a number which uniquely identifies the type of error or condition. Byte 2 is used to indicate the oncode and message associated with the error or condition. When bytes 3 and 4 are present, they are used to indicate which condition built-in functions may be required, and are copied into the flag byte in the current ONCA.

The error code is generated by the object program or library module, for software-detected PL/I conditions, and by the error handler itself for hardware-detected interrupts.

Enable Cells

Enable cells are held at a fixed offset in each DSA. They are a set of flags that correspond to the PL/I conditions whose enablement can be controlled by the programmer. The PL/I conditions referred to are as follows:

```
Bit 0   CHECK*
Bit 1   ZERODIVIDE
Bit 2   FIXEDOVERFLOW
Bit 3   SIZE
Bit 4   CONVERSION
Bit 5   OVERFLOW
Bit 6   UNDERFLOW
Bit 7   STRINGSIZE
Bit 8   STRINGRANGE
Bit 9   SUBSCRIPTRANGE
Bit 10  CHECK*
Bit 11  CHECK*
```

*See section on handling CHECK for details.

A flag is set to zero when the relevant condition is enabled. Two sets of these flags are held in each DSA - the block enable cells and the current enable cells. The block enable cells indicate the situation at the start of the procedure or block. The current enable cells show the position at the point in the program that is currently being executed. Having two sets simplifies resetting the flags when enablement is changed for a single statement. It also simplifies resetting of the enable cells should the statement result in entry to an on-unit or function reference, and simplifies return of control to a different part of the block because of a GOTO in the on-unit or function.

ONCBs (ON Control Blocks)

ON control blocks are used to address the on-unit and hold various information about

UNQUALIFIED CONDITIONS

1. A flag at the head of the DSA indicates that static ONCBs exist for that block.
2. The block and current enable cells indicate which of those conditions that are under programmer control are enabled at any given point in the program. Each such condition is represented by a single bit in each cell.
3. There is an on-cell for every ON-statement in the block. Each on-cell consists of a one-byte code identifying the condition, e.g., X'0A' (SUBSCRIPTRANGE). If the same condition appears more than once, previous on-cells are set to zero.
4. Static ONCBs are held contiguously in static storage, in the same order as the corresponding on-cells. They contain a code byte and flags that indicate such things as : whether SYSTEM was specified, whether SNAP was specified, whether the on-unit consists of a single GOTO statement, whether it is a null on-unit, etc. If there is an on-unit, its address is given in the second byte. (For GOTO-only on-units, the offset of the address of the label variable is given.)

QUALIFIED CONDITIONS

1. A flag at the head of the DSA indicates that dynamic ONCBs exist.
2. Dynamic ONCBs are set up during execution of each block in which qualified condition ON-statements occur. The last two words of a dynamic ONCB contain the same type of information as static ONCBs (described above, under 'Unqualified Conditions'), but use additional flags to indicate whether the condition is enabled and whether it is established. The second word contains qualifying information, such as the address of the FCB (for conditions such as ENDFILE, RECORD, TRANSMIT, KEY, etc.), or address of a symbol table (for ON CHECK on-units).
3. Dynamic ONCBs are chained together, the most recent being addressed from a fixed offset in the DSA. The last dynamic ONCB in the chain contains zero in its backchain field.

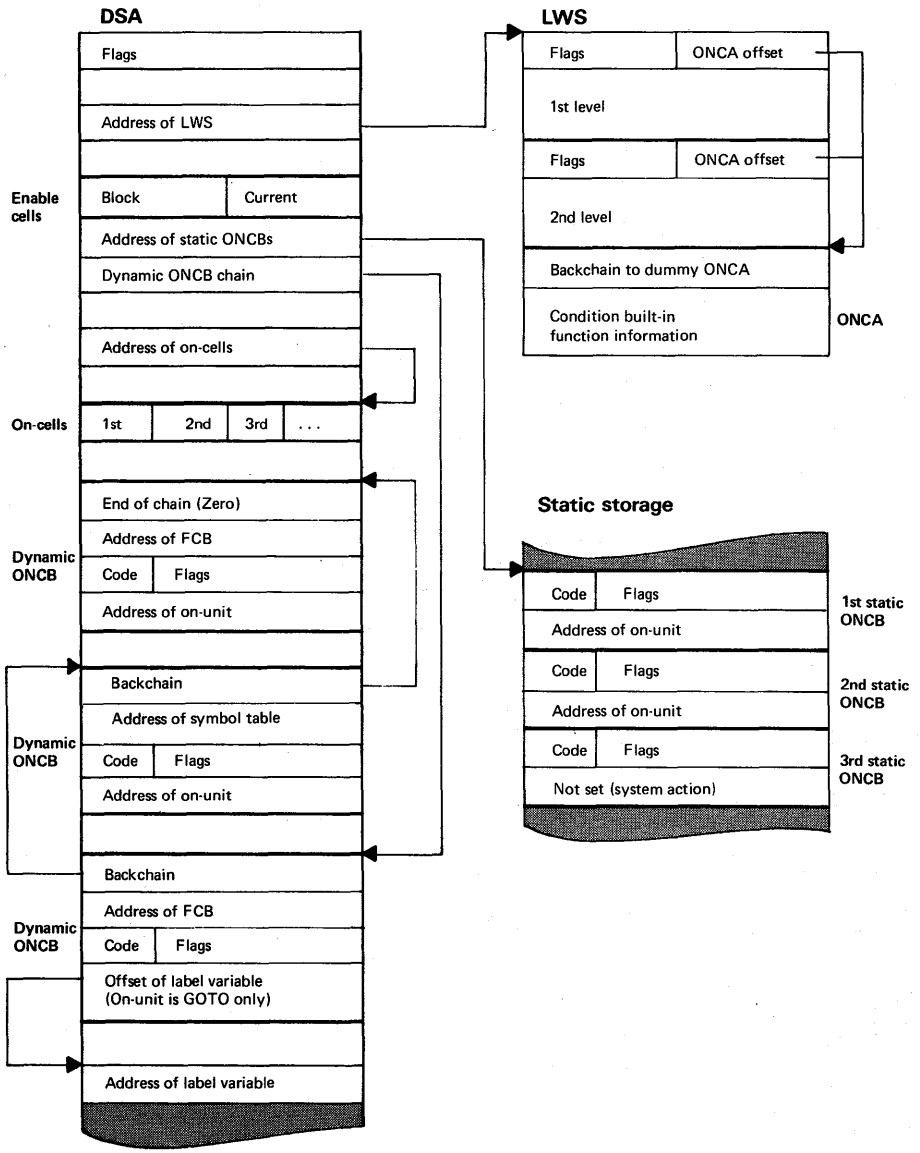


Figure 7.3. The principal fields used in error handling

the on-unit, for example, whether or not it is a null on-unit.

1. Dynamic ONCBs: Dynamic ON control blocks are held in dynamic storage and are used for conditions that need qualification - ENDFILE, ENDPAGE, KEY, NAME, RECORD, TRANSMIT, UNDEFINEDFILE, CHECK, and the CONDITION condition. The dynamic ONCBs for each block are chained together, and the address of the first dynamic ONCB is held at offset X'60'(96) in the DSA of each block. One ONCB is generated for each condition and qualifier regardless of the number of ON-statements in the block for that condition and qualifier. Dynamic ONCBs, but not static ONCBs, are used to test if there is a relevant established on-unit for the condition and qualifier.

2. Static ONCBs: Static ONCBs are held contiguously in static storage and refer to conditions that do not need qualification. A static ONCB is generated for every ON-statement in the block and holds the address of the associated on-unit if there is one. Each ONCB is associated with an oncell held in the DSA of the associated block. On-cells and ONCBs are held in the same order. The error-handling module searches the on-cells, counting how many on-cells it tests before the relevant on-cell is reached. The static ONCB is addressed by an offset from the address of the first static ONCB. The offset is calculated by multiplying the count of on-cells by eight, the length of a static ONCB. The address of the first static ONCB is held at offset X'5C'(92) in the DSA.

The addresses of the ONCBs are held at fixed offsets in the DSA. Dynamic ONCBs are chained together. Static ONCBs are held contiguously and can be found by incrementing the value of the first address until the correct ONCB is reached.

ONCA (ON Communications Area)

The ON communications area contains fields to hold the values, or address of the values, of any condition built-in functions that may be used. Flags are set to indicate which values are valid for the particular interrupt. The appropriate data is placed in the ONCA by compiled code or library modules. The on-code is not generated until required. Instead an error code is used. This gives greater flexibility in generating messages, as the error can, in certain circumstances, be defined more accurately than on-codes allow. On-codes are compatible with those used in previous PL/I compilers.

Dummy ONCA

ONCAs are chained together, so that a search can be made for the correct condition built-in function values through various levels of interrupt. The dummy ONCA, held in the program management area, acts as an effective end to the chain, and contains the default values for condition built-in functions.

Dummy DSA

The dummy DSA is not set-up exclusively for the use of the error-handling module. However, it plays an important part in error handling. After an interrupt, all existing DSAs are searched for an established on-unit that corresponds to the interrupt. The dummy DSA acts as an end in the chain that is searched. When the dummy DSA has been reached, standard system action is taken.

The dummy DSA is also used to avoid the need for providing special-case code when calculating the address of the interrupt, and the name of the entry point, when an interrupt has occurred in the main procedure.

Translate-and-Test Table

When an unqualified condition has been raised, the translate-and-test table in the TCA is used by a translate-and-test (TRT) instruction to test the on-cells and see if a relevant on-unit is established.

Executing ON and REVERT Statements

Executing ON and REVERT statements is essentially a matter of setting a flag, either on or off, in an on-cell or in a dynamic ONCB. The action depends on whether the associated PL/I condition is qualified or not, and hence whether an on-cell or a dynamic ONCB contains the flag.

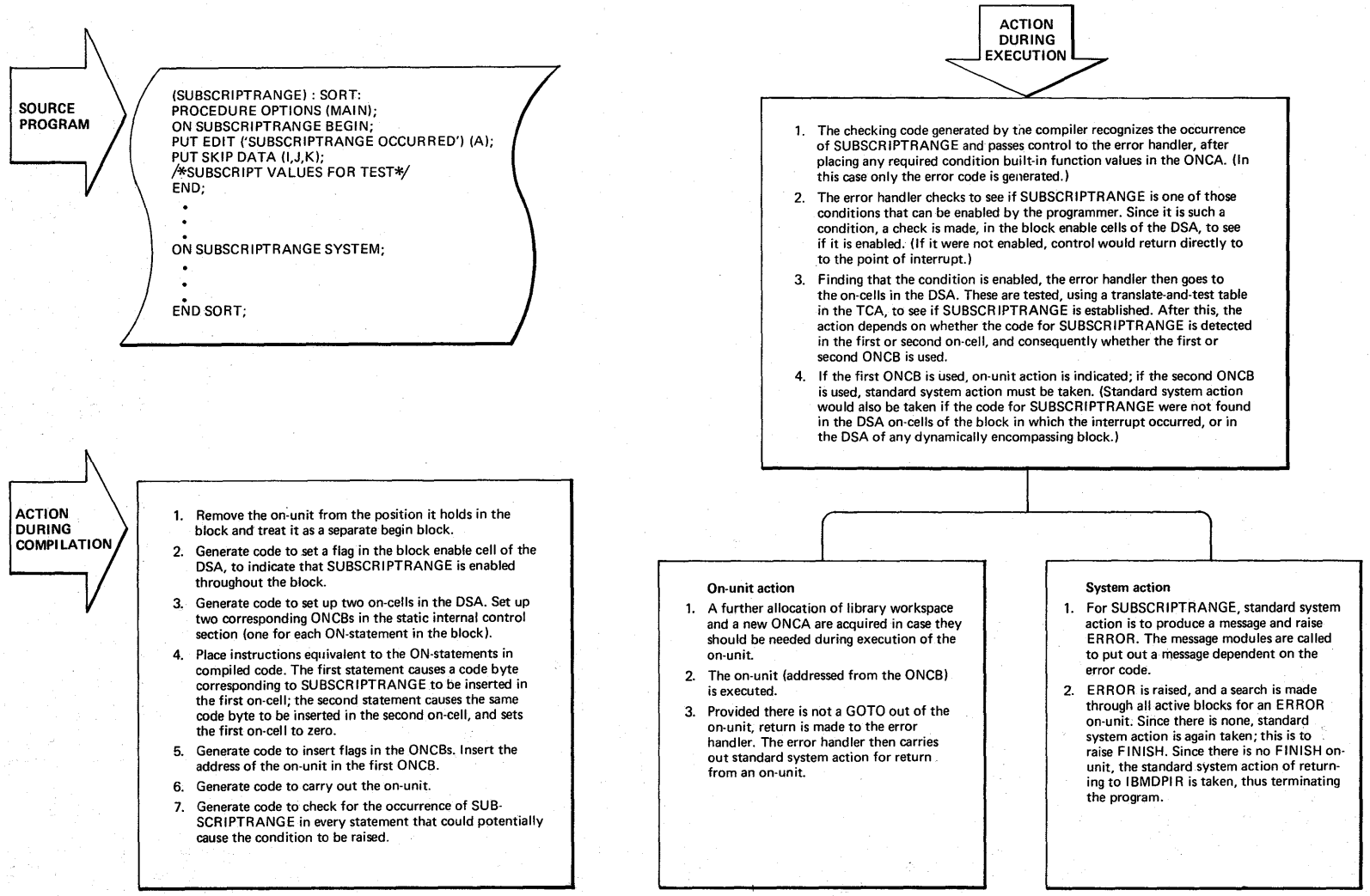


Figure 7.4. Example of error handling

Unqualified Conditions

For unqualified conditions, the ON-statement action is merely to set a flag on in an on-cell which is associated with that statement. If there is more than one ON-statement for the same condition in a block, the previous on-cells will be set to zero when second and subsequent on-cell flags are set on. The REVERT statement is executed by setting the flag in the latest on-cell to zero. The situation then reverts to that at the start of the block.

Qualified Conditions

On-cells are not generated for qualified conditions; instead, ONCBs are generated in the DSA of the block in which the ON-statement appears. When the ON-statement is executed, one of the ONCBs is associated with the condition. The ONCB has the qualifier and the address of the associated on-unit placed in it, and the 'established' flag set on. If a further ON-statement for the same condition with the same qualifier has to be executed, the address of the on-unit is changed. For a REVERT statement, the establishment bit is set off. This returns the situation to that at the beginning of the block.

IBMDERR-Error-Handling Module

The error-handling module, IBMDERR, handles three situations. These are as follows:

1. Hardware interrupts.
2. PL/I conditions detected by the object program.
3. Errors detected by the object program that are not directly related to PL/I conditions and which raise the ERROR condition.

All three situations are ultimately dealt with as PL/I conditions. For example, the FIXEDOVERFLOW condition would be raised when fixed point overflow occurs and causes a program check interrupt. Where there is no directly-applicable, PL/I condition (for instance after a data interrupt) a system message is printed and the ERROR condition is raised.

HARDWARE INTERRUPTS

The STXIT macro instruction, issued by IBMDPII during program initialization, specifies a save area and the address of a user exit routine to which control is to be passed after a program check interrupt. (All program check interrupts except "significance" and certain input/output are intercepted.) When a program check interrupt occurs, the supervisor saves the PSW and the contents of registers 0 through 15 at the time of interrupt in the special save area provided, and then passes control to the user exit routine.

The "exit routine" consists of two fullwords of machine instructions, in the TCA appendage, immediately preceding the interrupt save area. These instructions set up addressability and ensure that register 1 is pointing at the interrupt save area before branching to entry point A of the error handler, IBMDERR.

Before a program check interrupt can be handled by IBMDERR as a PL/I condition, action must be taken to prevent the system terminating the job should a further program check interrupt occur. The second word of the PSW (containing the interrupt address) is stored by IBMDERR in the register 14 slot of the save area which was current when the interrupt occurred. Registers 0 - 12 are also saved in this DSA. (Register 12 is saved in the field normally used for register 15. Registers 14 and 15 are saved later in IBMDERR's DSA.) IBMDERR then changes the address in the PSW in the save area to an address in IBMDERR. An EXIT macro instruction is then issued to indicate to the supervisor that the program check exit is finished. Control then passes via the supervisor to the address in IBMDERR that has been inserted in the PSW. As an EXIT macro has been issued, handling of the interrupt appears to the supervisor to be finished. The address, in the field in the TCA, to which control will pass after a program check interrupt is then changed to IBMBERRC. Should an interrupt now occur during the execution of IBMDERR, control will pass to IBMBERRC, which terminates the job with a DUMP macro. IBMDERR can now handle the interrupt. Having changed register 12 to the PL/I register 12 as previously saved, its first task is to generate a suitable error code that will equate the interrupt with a PL/I condition.

The floating point registers are saved in IBMDERR's DSA, if the interrupt is one corresponding to a PL/I condition, and control can then be passed to the main PL/I condition-handling routines described in the next section. There are, however, three

special cases that require further action. These are:

1. If the interrupt was floating point underflow, then the doubleword in which the floating point register which underflowed was stored is set to zero.

2. If fixed point, exponent or decimal overflow, or fixed-point divide has occurred, then this may correspond to the PL/I condition SIZE and not to FIXEDOVERFLOW or ZERODIVIDE. If this is the case, a flag will have to be set in the program check interrupt qualifier in the TCA. A test of this flag is therefore made and the necessary action taken.

3. If an operation exception occurs a branch is made to the operation interrupt analysis code in the program management area. A return code indicates whether the interrupt was caused by an attempt to execute a floating point instruction on a machine with no floating point hardware. The analysis code is set by IBMDPII as described in chapter 5.

SOFTWARE INTERRUPTS

When the main condition-handling logic is reached, an error code will have been generated to indicate the type of error or condition that has been raised. For program check interrupts, the code is produced by the error module itself. For errors or conditions detected by the object program, the object program sets up this code. When the object program has detected the error, this will, in some cases, correspond to a PL/I condition. However, there are certain errors (such as attempting to take the square root of a real negative number) that do not have directly-related, PL/I conditions. For PL/I conditions, a four- byte code is passed. For other errors, the code consists of only two bytes. In the second case, the first byte indicates which class of error has occurred (I/O, computational, etc.). In the first case, the first byte is the identifier of the PL/I condition being raised (the same identifier is used in on-cells).

The error-handling module checks the first byte of the code to see whether it is handling an error or a PL/I condition. If the code indicates an error, then the message module IBMDESM is loaded into a VDA and called. This module prints the relevant diagnostic message; a suitable four-byte code is then generated, and the ERROR condition is raised. The situation is then treated as a PL/I condition.

The second two bytes of the code passed when a PL/I condition has been raised indicate which on-unit built-in functions are relevant to the condition. If the condition is one that needs to be qualified, the qualifier is also passed.

When a PL/I condition code is passed, action depends on whether the condition is one of those that can be enabled or disabled by the programmer. If it is such a condition, a test is made in the current enable cells of the DSA. If the condition is not disabled, then a search for a relevant established on-unit must be made. If the condition is disabled, a return is made to the point of interrupt. If the condition is not qualified, then a search is made through the on-cells of all active blocks to find a match for the number in the first byte of the code passed to IBMDERR. This is done with a translate-and-test instruction using the TRT table in the TCA. When found, the offset of the located on-cell gives the offset of the associated ONCB. A test can then be made to determine the action to be taken.

If the condition is one that needs qualification, a search for an active matching ONCB is carried out through the chain of dynamic ONCBs held in each DSA. If the dummy DSA is reached without a match being found, then standard system action is taken. This action is defined in IBMDERR. (The CHECK condition, which may be either qualified or unqualified, is handled differently, as described later in this chapter.) When a matching active ONCB is found, tests are then made, as follows, on the flags in the ONCB.

Test 1. Is SNAP specified? If so, the message module IBMDESM is dynamically loaded and a SNAP message printed.

Test 2. Is SYSTEM specified? (This can occur when "ON condition SYSTEM" has been specified.) If SYSTEM is specified, then the action for system action specified in IBMDERR is taken.

Test 3. Does the on-unit consist only of a GOTO statement? If so, then the GOTO is executed without entering the on-unit. This saves the housekeeping involved in entering an on-unit.

Test 4. Is the on-unit a null on-unit? If so, then the action on a normal return from the on-unit is taken.

If none of these tests is positive, then it is necessary to enter the on-unit.

Before entering the on-unit, the following action must be taken. A new allocation of library workspace and ONCA must be initialized and its address put into the standard offset in the DSA of IBMDERR. This provides workspace for any further library modules that may be called. Tests must be made to see that the ONCA is correctly set-up for any built-in functions that may be used. The linkage to the error handler must also be altered to its original settings so that program check interrupts will cause entry to be made to the error handler by the entry point IBMBERRA rather than IBMBERRC. This ensures that the action specified by the PL/I program is taken if a program check interrupt occurs during the execution of an on-unit.

Normal return from the on-unit to IBMDERR is made by a branch on register 14. Depending on the condition, a return to the interrupted program is then made, or some special action may be taken. Five PL/I conditions cause action other than return to be taken.

1. If the condition was the ERROR condition, the FINISH condition is raised.

2. If the FINISH condition was raised by a STOP statement or the raising of ERROR, or by the normal termination of the main procedure, then a return code is set in the correct field of the TCA, and GOTO performed to the initialization routine IBMDPIR. If FINISH was signaled, then return is made to the point of interrupt.

3. If CONVERSION was raised, then a test is made in the ONCA, and if either ONSOURCE or ONCHAR has been accessed, control is passed to the address contained in the retry slot in the ONCA. The conversion is then attempted again. If the field has not been changed, then the ERROR condition is raised.

4. If ENDPAGE was raised, then a return code is set in register 15 to indicate than an on-unit has been entered.

5. If the condition was SUBSCRIPTRANGE, ERROR is raised.

RETURN TO POINT OF INTERRUPT

Software Interrupts

If the condition was one that was detected by compiled code, then a return to the point of interrupt is made by a branch on register 14.

Hardware Interrupts

For program check interrupts, the status of the program at the original point of interrupt has to be restored. This means that the contents of the system save area must be reset, so that they are identical with those saved after the original interrupt. (The PSW and the register values were saved in the interrupt DSA on entry to IBMDERR.)

The method used is as follows. The address that is to be branched to, after a program check interrupt, is changed from IBMBERRC to another point in IBMDERR. An interrupt is then caused, and the supervisor gains control. Consequently, the address in IBMDERR is reached with the address of the system save area in register 1. The contents of the save area and the PSW are then changed to those that were current after the original interrupt including the original value of register 12. The point of entry for program check interrupts is then reset to IBMBERRA. An EXIT macro is then issued, and return is made to the address in the PSW, which is that of the instruction following original interrupt.

The CHECK Condition

The CHECK condition has to be handled in a different manner to other conditions. This is because it can be used as a qualified or unqualified condition and its enablement is under programmer control.

The CHECK condition is disabled by default and is enabled by writing a CHECK prefix. It can be disabled for the duration of a statement or block by the NOCHECK prefix. Prefixes can take the form (CHECK) or (NOCHECK), or the form (CHECK(A,B)) or (NOCHECK(A,B)). When no name list is appended, the CHECK applies to all the relevant names in the program. An ON-statement may also be written as either ON

CHECK or ON CHECK(A,B). ON-statements are independent of prefixes and may be included in a block to which no prefix applies. A qualified on-unit can be used with an unqualified prefix and vice-versa.

Throughout this discussion, CHECK and NOCHECK without a name list are referred to as unqualified. CHECK or NOCHECK with a name list are referred to as qualified.

RAISING THE CHECK CONDITION

CHECK is normally raised by compiled code. This is done by inspecting the source program and generating calls to the error handler at appropriate points. As enablement is statically descendant, it is possible to tell during compilation at which points CHECK is enabled and consequently at which points the calls to the error handler have to be made. For GET DATA statements, however, there is no way of predetermining which items will be present in the input stream; when an item is input, therefore, the symbol table for that item is inspected to determine whether the CHECK condition may be enabled, and, if so, the error handler is called.

With the exception of the CHECK condition, all conditions whose enablement is under programmer control are unqualified conditions. Consequently, their enablement or disablement can be indicated by one bit in the enable cells. This is because there are only two possibilities. Either the condition is enabled or it is disabled. With CHECK, however, there are many possiblities, because CHECK may be enabled for some variables and disabled for others. Consequently, the enable cells are used in a different manner for the qualified CHECK condition, and the enablement of qualified CHECK for any particular name is given in an ONCB.

When the CHECK condition is raised, the error handler has to carry out the following tasks.

1. Test to see if CHECK is enabled. This involves a search along the static backchain to determine, for each block, first if qualified CHECK is enabled or disabled for the particular name for which CHECK was raised, and then if unqualified CHECK is enabled or disabled. (This test is carried out only if it is not known at compile-time that the CHECK condition is enabled.)

2. Search for a qualified established on-unit. This involves searching the dynamic backchain for a relevant dynamic ONCB.

3. If there is no qualified established on-unit, search for an unqualified established on-unit. This involves a further search of the dynamic backchain looking for appropriate on-cells.

4. If no established on-unit is found, take standard system action.

This process is illustrated in figure 7.5.

TESTING FOR ENABLEMENT

There are three bits that refer to CHECK in the enable cells; they have the following significance:

Bit 0
0 CHECK is enabled for certain items
1 CHECK is disabled

Bit 10 (only valid if bit 11 is set)
0 The unqualified prefix that applies is NOCHECK
1 The unqualified prefix that applies is CHECK

Bit 11
0 No unqualified prefix applies
1 An unqualified prefix applies

Bit 0 is referred to as the "any-CHECK" enablement bit, and bits 10 and 11 as the "unqualified CHECK enablement bits." Enablement and disablement of qualified CHECK is indicated in the flag bits of the ONCB.

The test for enablement begins by a test on the any-CHECK bit (bit 0) in the enable cell. If this is set to '1'B, control is immediately returned to the caller. If the bit is set on, a search is made for a relevant qualified ONCB in the DSA of the block in which the interrupt occurred; if such an ONCB is found, the enablement status is determined from it. If no such ONCB is found, the unqualified CHECK enablement bits are tested for unqualified enablement or disablement. If bit 11 is on, the enablement status is taken from bit 10. If bit 11 is not set, neither an unqualified CHECK nor an unqualified NOCHECK applies, and a further search must be made in the preceding DSA on the static backchain. If the dummy DSA is reached without any of the tests proving positive, CHECK is disabled.

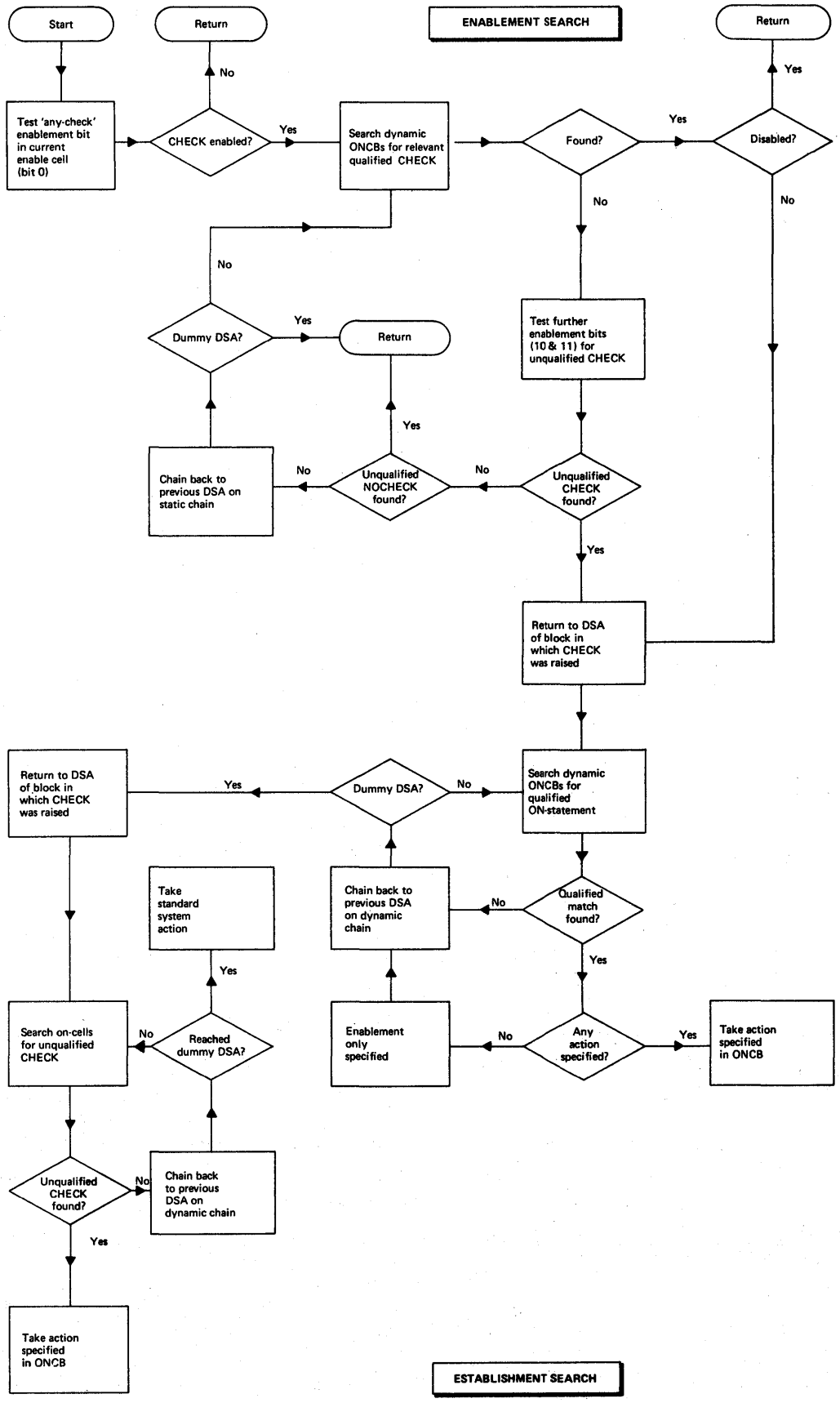


Figure 7.5. Handling the CHECK condition

SEARCHING FOR ESTABLISHED ON-UNITS

When it is known that CHECK is enabled, a search must be made for established on-units. This search is separate from the search for enablement. A return is first made to the DSA in which the interrupt occurred.

Two searches are made, the first for a qualified on-unit. The complete dynamic backchain is searched for relevant ONCBs. If one is not found, a search is made through the backchain for enable cells that indicate unqualified CHECK. If nothing is found, standard system action is taken.

Error Messages

The library module IBMDESM is called by the error handler to generate the system messages and find the on-code value; control is then passed to IBMDESN to finish the system message, or to generate the SNAP message. In order to save space, IBMDESN is overlaid on IBMDESM. No new DSA is acquired for IBMDESN.

Message Formats

System Messages: For non-PL/I conditions, system messages have the following form:

```
IBMxxxx 'ONCODE'= xxxx message text
[qualifier] IN STATEMENT xx AT/NEAR
OFFSET xxx IN PROCEDURE WITH ENTRY xxxx
```

The qualifier might, for example, consist of the file name. For PL/I conditions, the format of the message is much the same, but the name of the condition is also given. For example:

```
IBM4821 'ONCODE'= 3108 'FIXEDOVERFLOW'
CONDITION RAISED IN DECIMAL DIVIDE IN
STATEMENT 31 AT OFFSET 000A35 IN
PROCEDURE WITH ENTRY ZERNES
```

Snap Messages: If an on-unit contains both SNAP and SYSTEM, the resulting message is essentially the system message followed by the line

```
FROM (STATEMENT/OFFSET) xxx IN A (BEGIN
BLOCK/PROCEDURE WITH ENTRY xxx/A 'xxxx'
ON-UNIT)
```

which is repeated as many times as necessary to trace back to the main procedure. If an on-unit contains only SNAP, the message begins

```
'xxxxxxx' CONDITION RAISED [IN STATEMENT
xxx] (AT/NEAR) OFFSET xxx IN PROCEDURE
xxx
```

and continues as for a SNAP SYSTEM message.

The statement number is not always present in messages because the generation of execution-time statement numbers by the compiler is a compiler option. When statement numbers are generated, they are held on a block or procedure basis. For each block or procedure, a table in static storage relates each statement number to the offsets of the corresponding instructions in compiled code. A field addressed by compiled code gives the address of the relevant table.

The statement number is held in relation to its offset from the main entry point. Since the PL/I program need not have entered via this entry point, the offset is calculated independently from that given in the message. If the FLOW option is used, then additional information is printed out after every snap message. (See "The FLOW Option," later in this chapter.)

Interrupts in Library Modules

When an interrupt occurs in a library module, the system message does not give the offset from the start of the library module, but gives the statement number of the statement in which the library module was called and the offset of this statement from the entry point of the procedure block in which it is contained.

Identifying the Erroneous Statement

If the interrupt was a software interrupt in compiled code, the address will be the return address that was used by the BALR instruction when IBMDERR was called.

If the interrupt was a program check interrupt in compiled code, the address of the interrupt will have been moved from the old PSW and placed in the register 14 field by IBMDERR to simplify return to the point of interrupt.

If the interrupt was in a library module, the address required is the point in compiled code at which the library routine was entered. This will have been placed in the register 14 field when the library module was called.

Thus the address required to identify

the erroneous statement is always the address held in the register 14 field in the most recent compiled code DSA.

Finding the Address of the Entry Point of the Block

The address of the entry point of the block is found by chaining back along the DSAs to the DSA before the last compiled code DSA. The start of the chain is the DSA addressed by the current value of register 13. The address of the required entry point is held in the save area of this DSA as the branch register contents (offset X'C'). The existence of the dummy DSA ensures that there will always be a DSA before the one in which the interrupt occurred.

When the addresses of the entry point and the offending statement are known, the address of the block can be calculated and the statement number found. If the DSA in which the address of the link register was found is that of a procedure which was entered at its first entry point, then the offset already calculated will be the one required for the message. However, if the DSA containing the address of the interrupt was a begin block or on-unit, the offset for use on the statement number table is recalculated by using the exit address found above or taking from it the address of the main entry point, which is found in the statement number table. The offset is then calculated between the offending statement and the new address. This is necessary because the offsets given in the system message are taken from procedure entry points, whereas statement numbers are related to offsets in all blocks including begin blocks.

For snap messages, once the first procedure has been found and the appropriate message generated, the rest of the trace gives information about both procedures and on-units, and thus their DSAs are treated in the same way.

Ancillary Information

If the error was in I/O, then the address of the FCB of the file is passed to IBMDERR which stores it for IBMDESN to find the file name. Similarly, the address of the control section containing the condition name is passed to IBMDERR if the CONDITION condition is raised.

Message Text Modules

The message module IBMDESM calls on a number of message text modules to produce the relevant message. These modules consist essentially of the fixed message text portions of the message. The messages are held in groups. The groups are addressed from a table at the head of the module, and the messages in their turn are addressed by an offset from the start of each particular table in IBMDESM. The message required is determined from information in the error code. IBMDESN puts all error messages onto SYSPRINT provided that SYSPRINT has not been declared with unsuitable attributes. If it has been declared with unsuitable attributes, then the system messages go to the console operator, and the snap messages are ignored.

Dump Routines

A series of library modules are provided to implement the PLIDUMP facility. Module IBMDKDM is the dump bootstrap module which is part of the resident library. This loads and calls the transient dump control module IBMDKMR, which in turn loads and calls those modules required to carry out the dump options specified in the call to PLIDUMP. A number of transient modules are used to reduce the amount of storage required. The organization of these modules is shown in figure 7.6.

In order to ensure that as much information as possible is provided when a call to PLIDUMP is made, a special STXIT macro instruction is issued to intercept program check interrupts. When a program check interrupt occurs, an attempt is made to continue. If the interrupt occurs in a program called from the dump control module, that particular routine is abandoned and a return is made to the dump control module. Any further routines needed to complete the information specified in the options are then called. If the interrupt occurs in the trace or file modules, a hexadecimal dump is produced. If the interrupt occurs during IBMDKDD, the hexadecimal dump module, the job is stopped with a DOS system dump macro to ensure that a hexadecimal dump will be produced.

The dump control module IBMDKMR is divided into sections, and if an interrupt occurs in any of these sections, control is passed to a predefined address, at which point an attempt is made to continue with the next option. Processing then continues from that point. The dump modules are

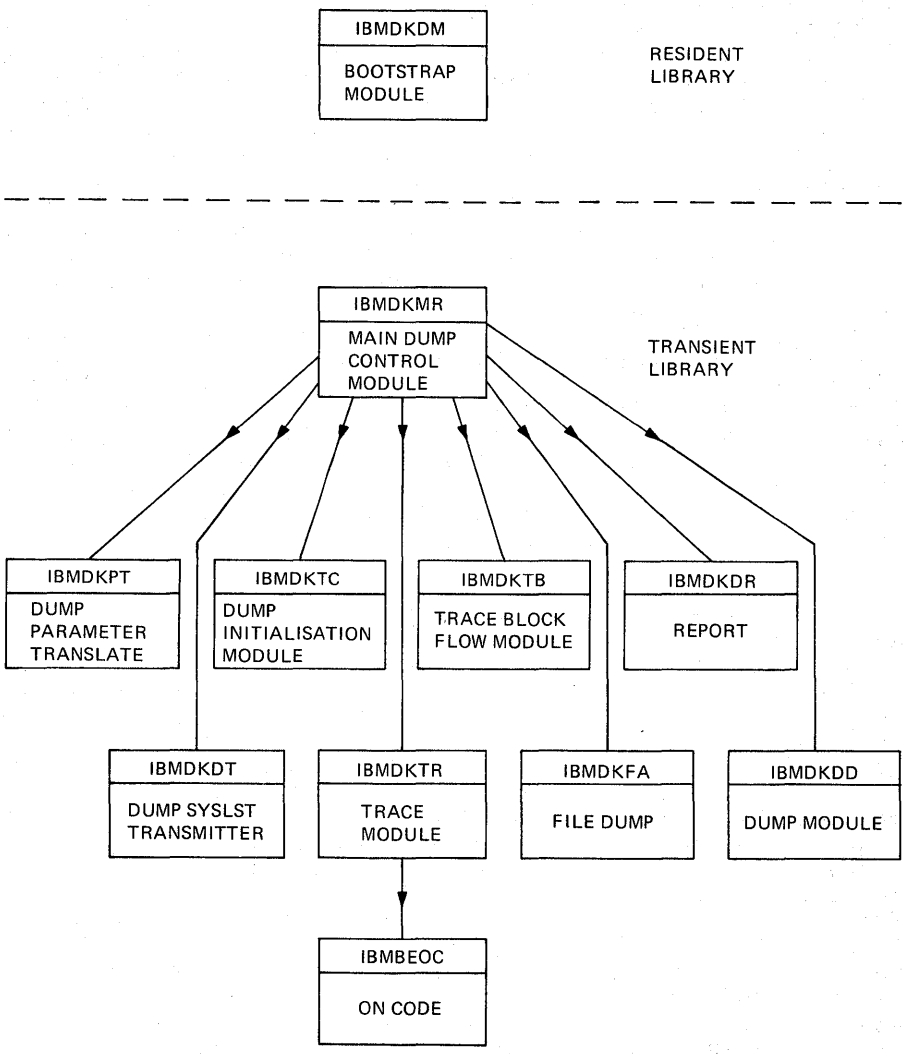


Figure 7.6. Interrelationship of dump routines

fully described in the publication DOS PL/I Transient Library Program Logic.

The dump control module acquires a VDA of the correct size for each module that it loads. When one module is completed it overlays the next module, adjusting the size of the VDA as necessary. Information is transmitted to SYSLST by the PLIDUMP transmitter IBMDKDT (this transmitter is also used for COUNT information). SYSLST must be assigned either to a printer or to a tape device.

Miscellaneous Error Routines

Two further library routines are used in certain error situations.

These are:

IBMDPEP Housekeeping error message module

IBMDPES Insufficient non-LIFO storage message module

Both routines are held in the transient library.

IBMDPEP: This module issues an appropriate message in three circumstances:

1. When there is no main procedure.
2. When there is insufficient space for the program management area at initialization.
3. When there is an interrupt in the error handler.

The first message is written on SYSPRINT. The last two messages are both written on the console, because SYSPRINT may not be able to function in the situations that are handled. This could be either because of overwriting of control blocks, if there is an interrupt in the error handler, or because there is insufficient room, if there is no storage available.

IBMDPES: If no dynamic storage is available, this module puts out a message using either SYSPRINT or the console message transmitter module, if available. If neither can be used, the message is written on the console using an EXCP macro instruction.

The FLOW and COUNT Options

The FLOW and COUNT options are used to provide information about which statements are executed in a particular run of a program. The FLOW option is used to maintain a trace of the most recently executed statements. The COUNT option is used to maintain a count of the number of times each statement is executed.

Both options are implemented by calling an interpretive library routine, IBMDEFL, at every point in a program where the flow of control may not be sequential. The library routine, IBMDEFL, analyzes the situation and updates tables to retain a record of the branches made. IBMDEFL is also called during program initialization to set up housekeeping information. Two transient library modules are used to interpret the tables set up by IBMDEFL and to put out the information. The routines are IBMDESN for the FLOW option, and IBMDEFC for the COUNT option.

The compiler generates the same executable code for both the COUNT and the FLOW option.

Points at which the flow of control may not be sequential are known as branch-in and branch-out points. For example, labeled statements and entry points are branch-in points, and GOTO statements are branch-out points. At branch-in and branch-out points the compiler places code that will call IBMDEFL. If the branches are taken, they are recorded. For COUNT they are recorded in a table known as the statement frequency count table. For FLOW, they are recorded in a table known as the flow statement table.

Use of Branching Information for FLOW

For the FLOW option, a list of the statement numbers at which branches were taken and a list of any changes of procedure is retained.

FLOW output consists simply of the list that is recorded by IBMDEFL and typically takes the form shown below.

```
        12 TO 18
        27 TO 35 IN SORTER
        76 TO 108 IN TESTER
        134 TO 77 IN SORTER
```

This indicates that the program branched from statement 12 to statement 18, then ran sequentially from 18 to 27. After statement 27 it branched to, or called, statement 35 in the procedure called SORTER. Control then ran sequentially to statement number 76, at which point it passed to statement number 108 in the procedure called TESTER. Control then ran sequentially from 108 to 134 and finally passed to statement 77 in SORTER.

Use of Branching Information for COUNT The COUNT option calculates the number of times each statement is executed by recording branch-in and branch-out points as they occur and analyzing them at the end of the program.

The formula used for calculating the number of times each statement is executed from the branch count is:

```
Cn=Cn-1+BIn-BOn-1
```

Where:

- Cn = the number of times the statement was executed.
- Cn-1 = the number of times the previous statement was executed.
- BIn = the number of times the statement was branched to.
- BOn-1 = the number of times the previous statement was branched from.

To retain the information, a count field is set up for every statement in the program, and branches-in and branches-out are recorded when they occur. Every time a branch-in is made, the count for the statement to which the branch is made is incremented by one. Every time a branch-out is made, the count for the statement after the branch-out is decremented by one. When the program ends, statements that have values other than zero mark the beginning and end of ranges of statements that have been executed the same number of times. The number of times the ranges of statements have been executed is calculated by adding the value in the count field to the sum of any preceding values. This process can be followed in figure 7.7.

PL/I PROCEDURE TO BE COUNTED

```
  1   COUNTIT:PROC OPTIONS (MAIN);
  2             DO I=1 TO 2;
  3                  PUT LIST (I);
  4             END;
  5           END COUNTIT;
```

In this procedure, the do-loop in statements 2 through 4 will be executed twice, and the other statements once. Statement 2 will be executed three times as a return is made at the end of the loop to test the value of I.

HISTORY OF THE STATEMENT FREQUENCY COUNT TABLE

After the branch-in to statement number 1, the table is set up with a value of 1 for the first statement and 0 for all others, thus:

statement number	1	2	3	4	5
branch count	1	0	0	0	0

After the branch-out at statement 4, the count of the next statement is decremented by one and the table becomes:

statement number	1	2	3	4	5
branch count	1	0	0	0	-1

After the branch-in at statement 2, the branch count for statement 2 is incremented by one and the table becomes:

statement number	1	2	3	4	5
branch count	1	1	0	0	-1

At statement 4, a further branch out is made and a return made to statement 2 to test the value of I. One is subtracted from the value of statement five makin the count -2 and one added to the count of statement 2 making it 2. Because I is greater than 2 a branch is made after the test to statement 5. This results in one being subtracted from the count for statement 3 and one being added to the count for statement 5. At the end of the program the table reads:

statement number	1	2	3	4	5
branch count	1	2	-1	0	-1

ANALYSIS OF THE STATEMENT FREQUENCY COUNT TABLE

A value known as the current count, which is initially set to zero, is added to the branch count for each statement in turn. The sum is the number of times the statement was executed; this value also becomes the current count.

statement number	current count	branch count	times executed
1	0	1	0+1= 1
2	1	2	1+2= 3
3	3	-1	3-1= 2
4	2	0	2+0= 2
5	2	-1	2-1= 1

Figure 7.7 How branch counts are used to calculate the number of times each statement is executed.

Special cases There are a number of special cases that require additional action, either by the compiler, or by IBMDEFL, or by both. These special cases arise for three reasons:

1. Branches can be caused by interrupts, but the points at which they will occur cannot be predicted during compilation. Consequently the compiler cannot place calls to IBMDEFL at these points.

2. Branches to labeled statements, can come from either the same block or a different block. Consequently the code generated by the compiler cannot be used to indicate whether a new block entry is required.

3. The algorithm used for the COUNT option is not effective for CALL statements and function references because the branch-in and branch-out are made to and from the same statement.

The first case is handled by IBMDEFL checking for the occurrence of an interrupt when it is called in situations where one could have occurred. The second case is handled by altering the GOTO code in the TCA so that it calls IBMDEFL to set appropriate flags when a GOTO out of block occurs. A test for the flags is made when the call to IBMDEFL for the branch-in at the labeled statement is made. The third case is predictable during compilation and is handled by the compiler setting up different code for branches-in to CALL statements and function references, and by IBMDEFL testing for such code.

IMPLEMENTATION OF FLOW AND COUNT

Tables Used by FLOW and COUNT

To enable it to retain FLOW and COUNT information, IBMDEFL sets up tables in dynamic storage. Details of their formats are shown in appendix B.

FLOW Option: FLOW information is retained in a table called the flow statement table. The flow statement table has three sections; a header section containing housekeeping information, a statement number section holding the numbers of statements that were branched to or from plus flags to indicate the type of entry, and a procedure names section containing the names of procedures and on-units to which branches are made. The length of the flow statement table is determined by the values given to "n" and "m" when the FLOW option is specified.

When all the spaces in the table for statement numbers or procedure names have been filled, the earliest entries are overwritten. The fields in the header section are used to indicate which is the next space available in the table.

The table is set up during program initialization and is addressed from the TCA.

COUNT Option: COUNT information is retained in tables called statement frequency count tables. The tables have a field for every statement. They are set up when an external procedure is entered. A table is needed for every external procedure because two external procedures can contain the same statement numbers.

Statement frequency count tables are chained together and addressed from the TCA appendage (the TIA). Two addresses are kept in the TIA, the address of the current statement frequency count table (that is the table that was last used) and the address of the statement frequency count table for the first procedure in the chain. Statement frequency count tables are associated with their matching external procedures by having the address of the static control section for the procedure placed at a fixed offset in the table. (A static control section is unique to an external procedure and its address can be easily accessed as it is addressed throughout compiled code by register three). The last statement frequency count table in the chain has its chaining field set to zero.

Interpreting the Flow Statement Table

Information from the flow statement table is interpreted by the message module IBMDESN or the PLIDUMP routines, and transmitted in the form of statement number pairs which are associated with the names of procedures or with on-unit condition types.

To extract the information, the message module must know from which points output in the statement number and procedure names section of the table output is to start. It must also be able to match the entries in the two sections of the table.

The starting points in both sections of the table are found by checking whether the dummy entry, inserted during program

initialization, has been overwritten. If the dummy entry has not been overwritten, the starting point is the first entry in that section of the table. If the dummy entry has been overwritten, the starting point will be the entry flagged as the next available entry. This is because the table is used cyclically, with the newest entry overwriting the oldest entry.

Statement numbers are matched with procedure names by comparing the number of procedure names with the number of statement number entries that are flagged as being associated with procedure name entries. If the two numbers are the same, the first procedure name will be associated with the first statement number that requires a procedure name. If there are more procedure names than statement numbers that require procedure names, the trace of procedures must be longer than the trace of statement numbers. Accordingly, the procedure names are put out without statement numbers until the point is reached where the number of procedure names left is the same as the number of statement numbers that require them. From that point on statement numbers and procedure names are put out together. If there are more statement numbers that require procedure names than there are procedure names, the trace of statement numbers must be longer than the trace of procedure names. The earliest statement numbers are put out without names and, where a procedure name is required, "UNKNOWN" is used. When the number of names required matches the number available, the procedure names are put out with the statement numbers.

Interpreting the statement frequency count tables

Module IBMDEFC is called at program termination to print count information. Output is tabular and printed four columns to a page. An entire page is built before transmission.

Output for a procedure begins with the procedure name. This is followed by the column headings: "FROM TO COUNT". The current count is initialized to zero and the first non-zero entry in the table is found. The associated statement number is then placed in the 'FROM' part of a temporary line and the value for the non-zero entry is added to the current count. The entries for the following statements are scanned until one with a non-zero count value is found. The number of the preceding statement is then placed in the 'TO' part of the line and the current count in the 'COUNT' part. This line is included in the page. The statement number found is then placed in the 'FROM' part of the temporary line and its branch count (which may be negative) is added to the current count. The scan of entries continues until another non-zero count is reached, and the process is repeated.

If the count for a range is zero, the line is not moved into the page but the two statement numbers are saved for separate printing. Whenever a line is moved into the page, checks are made for the end of a column and the end of the page. When the page is full it is transmitted.

The process is continued until the end of the table is reached.

The next table is then processed, until all procedures have been handled.

Finally, ranges of unexecuted statements are printed for each procedure.

Chapter 8: Record-Oriented Input/Output

Note on Terminology

Discussions of record-oriented input/output tend to become confusing because of the wide use that is made of the word "file" throughout the Disk Operating System. In the DOS usage, the word "file" means a collection of data stored on an external storage medium. Throughout this chapter, however, the term "data set" is used for this concept. "File" is used in its PL/I sense - the representation (within a PL/I program) of a data set.

Also used in this chapter are the terms record variable and key variable. These terms refer to the PL/I variables to which or from which data is moved. For example, in the statement:

```
READ FILE (X) INTO (Y) KEY (Z);
```

Y is the record variable, and Z is the key variable. The term transmission statement is used to cover READ, WRITE, LOCATE, and REWRITE statements. These three terms are not standard PL/I terminology, but they are used for convenience throughout this chapter.

Introduction

The DOS PL/I Optimizing Compiler uses the logical input/output control system (LIOCS) routines of DOS data management to implement record I/O. These routines offer facilities similar to, but not the same as, those of the PL/I language.

The LIOCS routines require that:

1. A define-the-file control block (DTF) is set up to describe and identify the data set.
2. OPEN and CLOSE macro instructions are issued to open and close the data set.
3. GET, PUT, READ, or WRITE macro instructions are issued to store or obtain a new record.

The LIOCS routines transmit the data one block at a time between the data management buffer and the external medium, but each separate macro instruction issued by the program results in only a single record being passed. When a transmission error occurs, or when the end-of-file is reached, the LIOCS routines either set flags indicating the error or branch to error handling or end-of-file routines that can be specified by the programmer.

The basic method used by the optimizing compiler to implement record I/O is to retain the source program information in a number of control blocks, and to pass these control blocks to PL/I library routines which interpret the information and carry out the necessary action by calling the data management LIOCS routines as required. The method is summarized below, and shown diagramatically in figure 8.1. Figure 8.16 shows the overall scheme in greater detail.

Summary of Record I/O Implementation

Compilation

During compilation the compiler sets up a number of control blocks that describe the file declaration, and the OPEN, CLOSE, and transmission statements. The compiler also generates code to complete these control blocks from execution-time information, and to pass their addresses to the PL/I library or LIOCS routines.

The compiler also determines which LIOCS routine will be used for each transmission statement, and generates an ESD record so that the appropriate routine will be link-edited.

If no environment options depend on execution time values, the compiler also acquires buffers, and completes the DTF and FCB. This reduces the work to be done during execution to little more than issuing the OPEN macro instruction. The process is referred to as optimization of the OPEN function in the compiler diagnostic messages.

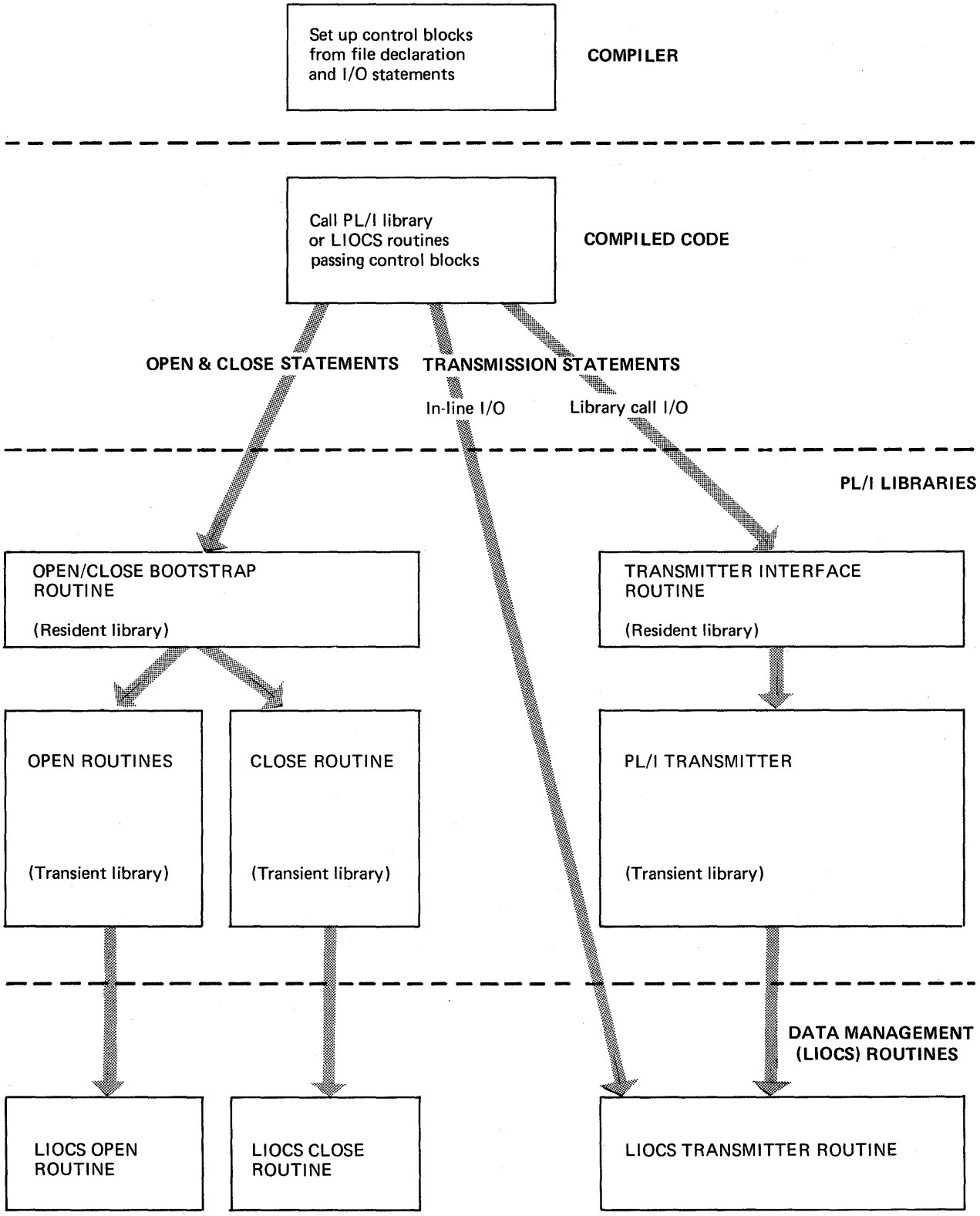


Figure 8.1. The principles used in handling record I/O statements

File type: Consecutive buffered		
Record type: F,FB		
Statement	**Record variable restrictions**	**ENVIRONMENT option requirements**
READ SET	None	None
READ INTO	Length known at compile time (max. length if a varying string)	RECSIZE known at compile time
WRITE FROM (fixed string)	Length known at compile time	RECSIZE known at compile time
WRITE FROM (varying string)		RECSIZE known at compile time SCALARVARYING option used
WRITE FROM Area *		RECSIZE known at compile time
LOCATE A	Maximum length known at compile time	RECSIZE known at compile time
Record type: U		
READ SET	None	Not BACKWARDS
READ INTO	Maximum length known at compile time	BLKSIZE known at compile time
WRITE FROM (fixed string)	Length known at compile time	BLKSIZE known at compile time
WRITE FROM (varying string)		BLKSIZE known at compile time SCALARVARYING option used
WRITE FROM (area *)		BLKSIZE known at compile time
LOCATE		BLKSIZE known at compile time
Notes: All statements must be found to be valid during compilation. File parameters or file variables are never handled by in-line code.		
* Including structures whose last element is an unsubscripted area.		

Figure 8.2. Conditions under which I/O statements are handled in-line

Data Set Organization	File Attributes			Access Methods
CONSECUTIVE	SEQUENTIAL	INPUT OUTPUT UPDATE	BUFFERED or UNBUFFERED	SAM
INDEXED	SEQUENTIAL	INPUT OUTPUT UPDATE	BUFFERED UNBUFFERED is ignored	ISAM
	DIRECT	INPUT UPDATE		ISAM
REGIONAL	SEQUENTIAL	INPUT OUTPUT UPDATE	BUFFERED or UNBUFFERED	DAM
	DIRECT	INPUT OUTPUT UPDATE		DAM
VSAM entry-sequenced	SEQUENTIAL	INPUT OUTPUT UPDATE	BUFFERED or UNBUFFERED	VSAM
VSAM key-sequenced	SEQUENTIAL	INPUT OUTPUT UPDATE	BUFFERED or UNBUFFERED	VSAM
	DIRECT	INPUT UPDATE	BUFFERED or UNBUFFERED	VSAM

Figure 8.3. Data management access methods for record-oriented transmission

Execution

PL/I record I/O statements are executed in the following manner:

OPEN - by a call to the open routines in the PL/I library.

CLOSE - by a call to the close routines in the PL/I library.

READ, WRITE, LOCATE, REWRITE - generally by a call to the PL/I library transmitter modules via an interface routine, IBMDRIO. The transmitters call the data management (LIOCS) routines. (This process is referred to "library-call I/O.")

On buffered consecutive files, most transmission statements are executed by a direct call from compiled code to the data management (LIOCS) routines. (This process is referred to as "in-line I/O.") Figure 8.2 shows the conditions under which I/O statements are handled in-line.

Implicit open - by manipulation of addresses so that all attempts to access the file when it is not open result in control being passed to the open routines in the PL/I libraries.

Implicit close - by the program termination routine checking for open files, and calling the PL/I library routine to close them.

File Type	Device Type	DTF Type
SEQUENTIAL BUFFERED CONSECUTIVE	Card	DTFCD
	Printer	DTFPR
	Tape	DTFMT
	Disk	DTFSD
	logical unit = SYSIPT, SYSLST, SYSPCH (F format only)	DTFDI
SEQUENTIAL UNBUFFERED CONSECUTIVE	Tape	DTFMT (work)
	Disk	DTFSD (work)
INDEXED	Disk	DTFIS
REGIONAL(1) AND (3)	Disk	DTFDA
Note: DTFs are not set up for VSAM files		

Figure 8.4. Type of DTF set up for different PL/I file types

Access Method and DTF Type

The access method used for different PL/I file types is shown in figure 8.3. The DTF used for different file types is shown in figure 8.4.

Compiler Output for Record I/O

The output of the compiler for record I/O is subdivided below according to the statement type in the source program. Figure 8.5 shows the control blocks generated for each statement type, and the relationship between these control blocks.

File Declaration

For every file declaration, except those with ENV(VSAM), a define-the-file control block (DTF), a file control block (FCB), an environment control block (ENVB), and a field to contain the filename and the length of the filename are set up. All these items are held in static internal storage for INTERNAL files, and in a separate control section for external files.

For a file declared with ENV(VSAM) a DTF is not set up. Instead an ACB (access method control block) is used which is generated by the library routines during execution. The FCB, ENVB and filename fields are set up as for other files. If a file that is declared with ENV(INDEXED) is used to access a VSAM key sequenced data set, the VSAM interface is used.

The DTF is required by DOS data management. There are nine types of DTF that can be used by the compiler. The type used depends on the file and device type, as shown in figure 8.4. Full details of the DTF are given in the publication DOS Supervisor and Input/Output Macro Instructions.

Where there are no environment options that depend on execution time values the compiler also acquires buffers and fills in the DTF fields with buffer addresses. The buffers are held within the file control section for external files and within the static internal control section for internal files. The compiler also does any necessary checking on items such as blocksize. If the declaration is invalid the compiler generates a message and makes no attempt to optimize the operation.

When the operation is optimized a flag is set in the FCB.

The FCB is a control block that is used as a central addressing area for file information. It holds the addresses of the DTF, and the ENVB, the filename and filename length. For VSAM files the

address of the ACB (access method control block) is set when the file is opened and the block has been generated. It also holds a mask indicating which statements are valid for the file. The format of the FCB is shown in appendix B.

Both the FCB and DTF are made as near complete as possible during compilation; those values which are not available until execution time are added to the blocks during the execution of the open routines. These values are derived from the ENVB and the open control block OCB (see below).

The ENVB contains the addresses of such items in the environment options as can be declared with variable values. The format of the ENVB is given in appendix B.

The filename and filename length are used by the error message modules when the name of the file is needed for an error message.

The OPEN Statement

For an OPEN statement, the compiler generates a call to the library open/close bootstrap routine, IBMDOCL; if the attribute input or output has been used in the OPEN statement, the compiler also generates an open control block (OCB). The OCB indicates whether the input or the output attribute was used in the statement. The OCB is held in static internal storage

For each file to be opened, the following information is passed to the open/close bootstrap routine:

1. The address of the FCB.
2. The address of the OCB (or zero, if no OCB exists).
3. The address of the TITLE, if specified.

Transmission Statements

The code and control blocks generated for a transmission statement depend on whether it will be handled by an in-line call to the LIOCS routines or by a call to the PL/I library transmitter.

In-line record I/O statements: the compiler generates a call to an LIOCS routine, which uses information in the DTF as a parameter list. Compiled code addresses the LIOCS routine through a field in the DTF. The DTF is addressed from a field in the FCB.

For an in-line call, code may also be generated either to move the data to the record variable or to set a pointer to the data, and to check whether the transmission has been successful.

The code and control blocks generated for an in-line record I/O statement are shown in figure 8.6.

Library-call record I/O statements: the compiler generates a call to the PL/I transmitter module, IBMDRIO. IBMDRIO has the following parameter list passed to it:

Address of FCB
Address of request control block (RCB)
Address of record descriptor (RD)); or, address ignore factor; or, address at which to set pointer
Address of key descriptor (KD) ; or zero if no key descriptor
Address of event variable (EV); or, zero if no event variable
Abnormal locate return address (locate statements only)

The FCB is generated from the file declaration and is described above. The remainder of the control blocks in each parameter list are generated for the transmission statement.

The request control block (RCB) defines the statement type. It consists of two words. The first word is a fullword of flags that define the statement types, indicating whether the statement is READ SET, READ INTO, WRITE FROM, etc. The second word is a machine language instruction that will be executed by IBMDRIO. (For exact format, see appendix B.) The RCB is set up in static internal storage.

A branch instruction is placed in the second word of the RCB if, during compilation, the statement type can be validated. A direct branch to the transmitter will then occur during execution. If, however, the statement type cannot be checked during compilation, or if it is invalid, a test-under-mask instruction is placed in the RCB. The check of statement validity will then be made during execution, using the flags in the FCB which indicate the valid statements for the file.

All transmission statements can be checked for validity during compilation except for statements on unbuffered consecutive files, file parameters, and file variables. Unbuffered consecutive files can be opened for either INPUT or

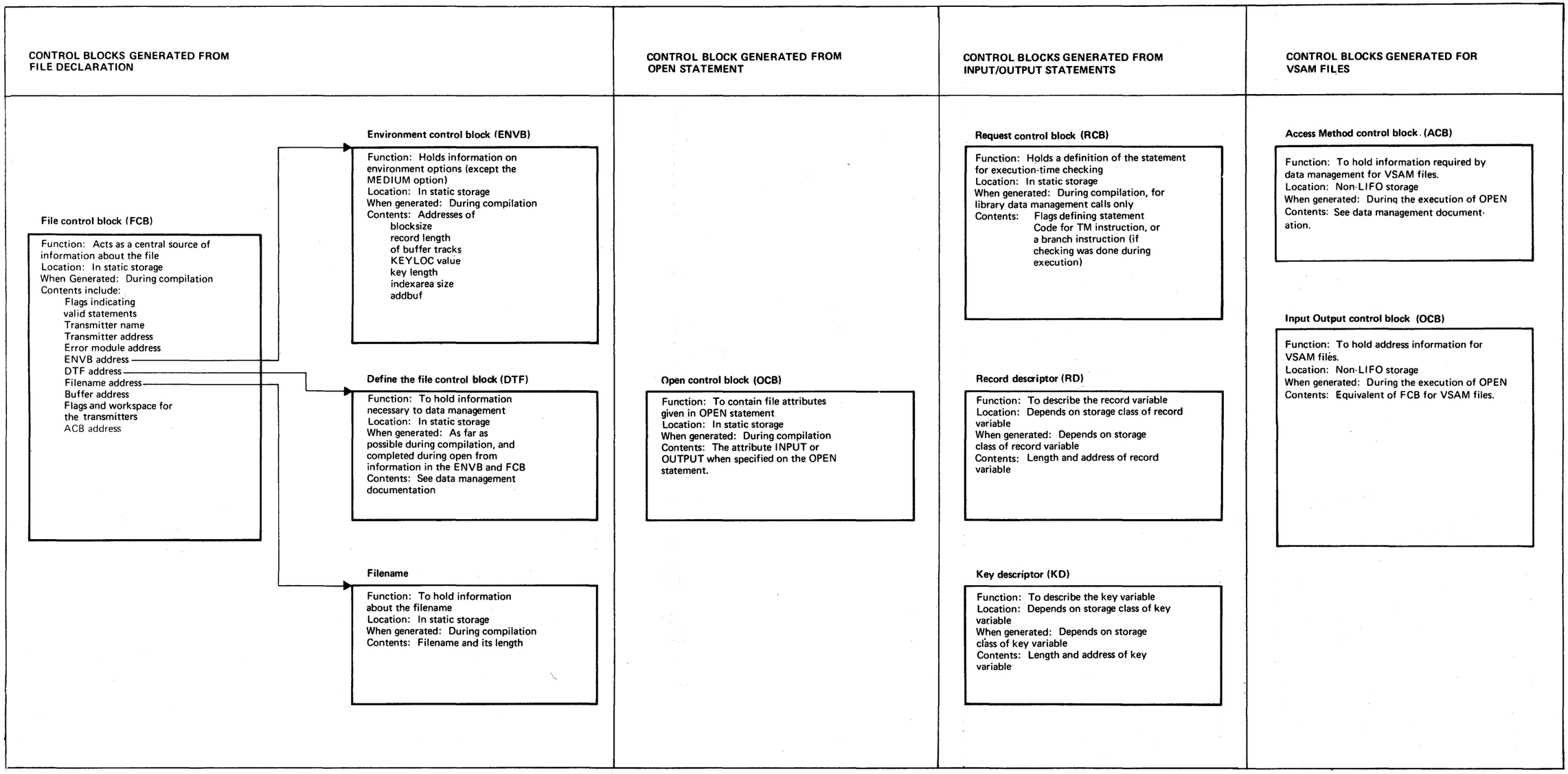


Figure 8.5. Control blocks used in record I/O

SOURCE LISTING

```
STMT

  1  EXAMPLE:PROC OPTIONS (MAIN);
  2            DCL LINE FILE RECORD INPUT
                   ENV(FB,RECSIZE(80),BLKSIZE(400),MEDIUM(SYSIPT,2540)),
                   CARD CHAR (80);
  3            READ FILE (LINE) INTO (CARD);
  4             END;
```

STATIC INTERNAL STORAGE MAP

```
000000  00000008              PROGRAM ADCON
000004  0000005E              PROGRAM ADCON
000008  0000005E              PROGRAM ADCON
00000C  00000000              A..IBMBOCLA
000010  00000000              A..IBMBOCLC
000014  00000000              A..IBMBRIOA
000018  00000000              A..IBMBRIOD
00001C  00000000              A..STATIC
000020  91E091E0              CONSTANT
000024  00000000              A..FCB
000028  000000940000005E      COMPILER LABEL CL.3
```

STATIC EXTERNAL CSECTS

```
000000  9100000000000000      FCB
        0000000000000000
        0000008800000090
        000000C000000000
        D8E9000002110100
        8880202002000000
        0190000000000050
        000000F8000000F8
        00000000001C0000
        00000000000000F8
        0000028800000000
        0000000000000000
        0000000000000000
        0000000000000000
        0000000000000000
        0004D3C9D5C50000
000090  C0000000010000B0      ENVB
        010000B4020000B8
        020000B8020000B8
        020000B8020000B8
0000B0  0000800000000001      DTF (CONSTANT PART)
0000B8  000000E0              DTF (VARIABLE PART)
0000BC  0000000000000000      DTF (CONSTANT PART)
        02050202
0000C8  00000288              DTF (VARIABLE PART)
0000CC  00000000              DTF (CONSTANT PART)
0000D0  020000F8              DTF (VARIABLE PART)
0000D4  2000019041BE0000      DTF (CONSTANT PART)
        470000000000
```

OBJECT LISTING

```
* STATEMENT NUMBER  3
000062  18 72                    LR    7,2          Set R2 as program base
000064  58 90 3 024               L     9,36(0,3)    Place address of FCB in R9
000068  18 29                    LR    2,9          Place address of FCB in R2
00006A  58 10 2 018               L     1,24(0,2)    Place address of DTF in R1
00006E  58 80 2 05C               L     8,92(0,2)    Pick up deblocking words in DTF
000072  58 B0 8 000               L     11,0(0,8)    Place previous record address in R11
000076  5A B0 8 004               A     11,4(0,8)    Add record length to old address
00007A  50 B0 8 000               ST    11,0(0,8)    Store in current record address
00007E  59 B0 8 008               C     11,8(0,8)    Test if current record address is in buffer
000082  47 D0 7 030               BNH   CL.2         If it is branch around LIOCS call
000086  41 80 3 028               LA    8,40(0,3)    Pick up end-of-file address from static storage
00008A  58 F0 1 010               L     15,16(0,1)   Call LIOCS routine to get new buffer
00008E  45 E0 F 008               BAL   14,8(0,15)
000092                  CL.2      EQU   *
000092  D2 4F D 0A8 B 000        MVC   CARD(80),0(11) Move record into record variable (CARD)
000098                  CL.3      EQU   *
000098  91 80 2 02C               TM    44(2),X'80'  Test for errors
00009C  47 E0 7 044               BNO   CL.4         Branch if no errors
0000A0  58 F0 3 018               L     15,A..IBMBRIOD If errors call record I/O bootstrap routine
0000A4  05 EF                     BALR  14,15
0000A6                  CL.4      EQU   *
0000A6  18 27                     LR    2,7          Restore R2 as program base
```

Figure 8.6. Annotated list showing record I/O statements handled by in-line code

```
                SOURCE LISTING

STMT

  1     EXAMPLE:PROC OPTIONS (MAIN);
  2                       DCL LINE FILE UNBUFFERED INPUT RECORD
                              ENV(F,RECSIZE(80),MEDIUM(SYS001,2311)),
                          CARD CHAR (80);
  3                     READ FILE (LINE) INTO (CARD);

  4             END;

    * PROLOGUE BASE
    00005E  D2 07 D 0A8 3 028          MVC   168(8,13),40(3)   move skeleton record descriptor to DSA
    000064  41 60 D 080                LA    6,CARD            complete descriptor with address
    000068  50 60 D 0A8                ST    6,168(0,13)       of record variable
    00006C  05 20                      BALR  2,0               set R2 as program base

    * PROCEDURE BASE

    * STATEMENT NUMBER  3
    00006E  41 90 D 0A8                LA    9,168(0,13)       pick up address of record descriptor
    000072  50 90 3 040                ST    9,64(0,3)         place in argument list
    000076  41 10 3 038                LA    1,56(0,3)         point R1 at argument list
    00007A  58 F0 3 014                L     15,A..IBMBRIOA    branch and link to IBMDRIO
    00007E  05 EF                      BALR  14,15

                        STATIC INTERNAL STORAGE MAP

    000000  00000008          PROGRAM ADCON
    000004  0000005E          PROGRAM ADCON
    000008  0000006E          PROGRAM ADCON
    00000C  00000000          A..IBMBDCLA
    000010  00000000          A..IBMBDCLC
    000014  00000000          A..IBMBRIOA
    000018  00000000          A..IBMBPGOA
    00001C  00000000          A..STATIC
    000020  0080000007FF0000  CONSTANT     request control block
    000028  0000000000000050  RECORD DESCRIPTOR
    000030  91E091E0          CONSTANT
    000034  00000000          A..FCB
    000038  00000000          A..FCB
    00003C  00000020          A..CONSTANT
    000040  00000000          A..RD
    000044  00000000          A..NULL ARGUMENT
    000048  00000000          A..NULL ARGUMENT
    00004C  80000000          A..NULL ARGUMENT

                        STATIC EXTERNAL CSECTS

    000000  1188000000000000  FCB
            0000000000000000
            0000008800000090
            000000B8C0000000
            C3E9000002110200
            8080000002000000
            0000000000000000
            0000000000000000
            0000000000000000
            0000000000000000
            0000000000000000
            0000000000000600
            0000000000000000
            0000000000000000
            0000000000000000
            0000000000000000
            0004D3C9D5C50000
    000090  4000000002000080  ENVB
            0100008402000080
            0200008002000080
                                    0200008002000080
                            000080  0000920400000101  DTF (CONSTANT PART)
                            000088  00000100          DTF (VARIABLE PART)
                            0000BC  0000000000000000  DTF (CONSTANT PART)
                                    2060D3C9D5C54040
                                    40000E2900000000
                                    0000000900640000
                                    0000000000000000
                                    0000000000000FF00
                                    0000006420000CA4
                                    A000CCAC
                            0000F8  070000EE          DTF (VARIABLE PART)
                            0000FC  40000006          DTF (CONSTANT PART)
                            000100  310000F0          DTF (VARIABLE PART)
                            000104  40000005          DTF (CONSTANT PART)
                            000108  08000108          DTF (VARIABLE PART)
                            00010C  000000000300D278  DTF (CONSTANT PART)
                                    2000000105000280
                                    20000001
                            000120  310000F0          DTF (VARIABLE PART)
                            000124  40000005          DTF (CONSTANT PART)
                            000128  08000128          DTF (VARIABLE PART)
                            00012C  00000000          DTF (CONSTANT PART)
                            000130  1E000148          DTF (VARIABLE PART)
                            000134  30000008          DTF (CONSTANT PART)
                            000138  12000148          DTF (VARIABLE PART)
                            00013C  0000000800000000  DTF (CONSTANT PART)
                                    01000000
```

Figure 8.7. Annotated object program showing record I/O statements handled by library subroutines

OUTPUT in the OPEN statement and, consequently, the statement validity cannot be determined until the file is opened. With file parameters and file variables, it is impossible to know which file will be referred to, and consequently the validity of statements using file parameters or file variables cannot be determined during compilation.

The format of the RCB is given in appendix B.

The record descriptor (RD) contains the address, length and type of the record variable. It is generated only if a record variable is required. (For exact format, see appendix B.)

The key descriptor (KD) contains the address and length of the key variable. It is generated only if a key variable is used. (For exact format, see appendix B.)

If the record variable or the key variable is STATIC INTERNAL, a complete RD or KD is set up and placed in static internal storage during compilation. In most other circumstances, a skeleton RD or KD will be set up which will be completed during execution by the inclusion of the address. The completed descriptor may be moved into temporary storage. In certain conditions, no skeleton is produced: the complete descriptor is built in temporary storage by compiled code.

The event variable (EV) contains information about the event that has been associated with the event I/O statement. (For exact format, see appendix B.) The implementation of event I/O is covered briefly at the end of this chapter, and more fully in chapter 11 under the heading "The WAIT Statement."

The abnormal-locate return address is used only for LOCATE statements. It is the address to which control will be passed if an error is detected in a locate statement and a normal return is made after execution of the on-unit. The abnormal-locate return address is usually the start of the next statement.

The code and control blocks generated for a transmission statement using a library call to the LIOCS routines are shown in figure 8.7.

CLOSE Statements

For CLOSE statements, the compiler generates a call to the open/close bootstrap routine, IBMDOCL, passing to it the address of the FCB, and, if required, flags indicating the presence of the LEAVE or UNLOAD option.

Library Routines in Record I/O

Because the amount of code involved in implementing a record I/O statement is quite large and would be duplicated for each similar record I/O statement, record I/O is handled mainly by PL/I library routines. The work done by library routines is summarized in figure 8.12.

Type of Library Modules Used

The library modules used by the compiler can conveniently be considered under three headings:

1. Open and close modules - called to open and close the files.

2. Transmitter modules - called to transmit data by calling the LIOCS routines. The PL/I transmitters hold the error and end-of-file routines for both library and in-line LIOCS calls.

   For consecutive buffered files, the error and end-of-file routines are provided as a separate library module, IBMDRRR. This module is loaded when the file is opened, and can be considered as part of the transmitter.

3. Error routines - used, when PL/I conditions occur, to handle housekeeping problems and calls to the error handler, IBMDERR.

   The routines involved are shown in figures 8.8, 8.9, and 8.10 respectively. Their interrelationship is shown in figure 8.11.

Because of their length, the major modules are held in the transient library, and are loaded and called when required by small resident modules.

The open and close routines are loaded by the open/close bootstrap routine, IBMDOCL. Transmitters are loaded by the transient open routines, and are called via the resident transmitter interface module, IBMDRIO. The error routines are loaded by bootstrap entry points in IBMDOCN. The module IBMDOCN contains the open/close parameter list and is seperated from IBMDOCL to improve overlong performance.

Resident library bootstrap routine IBMDOCL - loads and calls appropriate transient module	
Entry point	Function
IBMBOCLA	Explicit open
IBMBOCLB	Implicit open for library-call I/O
IBMBOCLC	Explicit close
IBMBOCLD	Implicit close
IBMBOCLG	Implicit open for in-line I/O
Transient library open and close modules	
Name	Function
IBMDOPM	Open stage 1 consecutive unbuffered files
IBMDOPP	Open stage 1 consecutive buffered files
IBMDOPS	Open stage 1 stream files
IBMDOPX	Open stage 1 regional and indexed files
IBMDOPO	Open stage 2 consecutive buffered files acquire buffers initialize DTF except for disk and tape
IBMDOPT	Open stage 2 stream files acquire buffers initialize DTF except for disk and tape
IBMDOPY	Open stage 2 regional and indexed files acquire buffers and initialize DTFDA
IBMDOPU	Open stage 3 consecutive buffered and stream files initialize DTF for disk or tape
IBMDOPZ	Open stage 3 regional and indexed files acquire buffers and initialize DFTIS
IBMDOPV	Open for VSAM files
IBMDOCA	Close files
IBMDOCV	Close for VSAM files

Figure 8.8. PL/I resident and transient library OPEN and CLOSE routines

Opening a File Explicitly

For an explicit open, a call is made to the resident library module IBMDOCL to open the file. This routine, known as the open/close bootstrap, is called once for every file that requires opening and is passed the address of the FCB, the address of the OCB, if one is required, and the address if the string locator for the title option, if the TITLE option is being used. IBMDOCL then calls one of five transient open modules depending on the file type.

For VSAM files the compiler places the address of IBMDOPV in the FCB and consequently IBMDOPV is called by IBMDOCL. IBMDOPV acquires space for an ACB (access control block), IOCB (input/output control block) and an RPL (request parameter list) and then creates an ACB using a GENCB macro instruction. IBMDOPV also sets fields in the FCB so that the correct error module will be called, and loads the appropriate transmitter.

For files other than VSAM there are four groups of modules, one group for consecutive buffered files, one group for stream files, one group for regional and indexed files and a single module for consecutive buffered files. The module called by IBMDOCL is the major module in the group. It has the job of issuing the open macro instruction, loading the transmitter if necessary, setting up the ERROPT and EOFADDR fields in the FCB and handling TITLE, PAGESIZE and any repositioning options such as REWIND. If the buffer space has been allocated during compilation no further action will be necessary. However, if buffers are required and the DYNBUFF option, variable environment options, or an invalid declaration has prevented buffer space being acquired during compilation, further action is necessary. The first transient modules then call further transient modules in the group to complete the DTF and get buffer space. These further modules are overlayed on the high address end of the first module. For certain file types it is

Resident library interface routines IBMDRIO/IBMDOCN	
Entry point	**Function**
IBMBRIOA	Test statement validity and call transmitter
IBMBOCNB	Bootstrap entry point to load and call error or endfile module on first error
IBMBRIOC	Call error handler with invalid statement code
IBMBRIOD	Entry from in-line code when errors detected. Branches to RIOB

Transient library transmitter modules IBMDRxx			
Data set organization	**File attributes**		**Transmitter module**
REGIONAL(1)	SEQUENTIAL	unbuffered output F-format	IBMDRAY
		buffered output F-format	IBMDRAZ
		unbuffered input/update F-format	IBMDRBZ
		buffered input/update F-format	IBMDRBW
	DIRECT	F-format	IBMDRDZ
REGIONAL(3)	SEQUENTIAL	unbuffered output F/U-format	IBMDRAW
		buffered output F/U-format	IBMDRAX
		unbuffered input/update F/U-format	IBMDRBY
		buffered input/update F/U-format	IBMDRBX
	DIRECT	F/U-format	IBMDRDY
CONSECUTIVE	SEQUENTIAL	unbuffered U-format	IBMDRCY
		unbuffered F-format	IBMDRCZ
		buffered U-format	IBMDRRX
		buffered V-format	IBMDRRY
		buffered F-format	IBMDRRZ
		associated file f-format	IBMDRRW
		associated file v-format	IBMDRRV
		associated file u-format	IBMDRRU
		OMR file F-format	IBMDRRT
		error and end-of-file exit module	IBMDRRR
INDEXED	SEQUENTIAL	input/update F-format	IBMDRJZ
		output F-format	IBMDRLZ
	DIRECT	input/update F-format	IBMDRKZ
VSAM (entry sequenced)	SEQUENTIAL	buffered/unbuffered input/output/update	IBMDRVZ
VSAM (key-sequenced)	DIRECT buffered/unbuffered input/update		IBMDRVR
	SEQUENTIAL buffered/unbuffered input/update		IBMDRVS
	SEQUENTIAL buffered/unbuffered output		IBMDRVT

Figure 8.9. Record I/O transmitters and their associated file types

Name	Function	Access method
ENDFILE module		
IBMDREF	Calling error handler for ENDFILE condition if general error module has not been loaded.	Any
General error modules		
IBMDREZ	Calling error handler, for buffered consecutive files	SAM
IBMDREX	Calling error handler, for indexed files	ISAM
IBMDREY	Calling error handler, for regional files and unbuffered consecutive files	DAM SAM (workfiles)
IBMDREV	Calling error handler for all errors (VSAM files)	VSAM

Figure 8.10. PL/I transient library error modules

necessary to call a third module which is overlayed on the second. See figure 8.11 for the interrelationship of these modules. After the second and, possibly, third module has been exectued a return is made to the first transient module to issue the OPEN macro instruction and load the library transmitter. The first module returns via the open/close bootstrap to compiled code.

Space for the open modules and for buffers is acquired in non-LIFO storage. IBMDOCL acquires 2K bytes for the transient open routines and, if buffer space is required and the length of buffers was known at compile time, also acquires sufficient storage for buffers.

The transmitter is only loaded if it is not already in main storage. (A test is made on the chain of loaded modules to see if it is.) If the transmitter is not already loaded it is overlayed on the high address end of the first transient module. It is moved down until the end of the transmitter is contiguous with the high address end of the 2K acquired for the transient modules. The transmitter is then contiguous with any buffer space that may have been acquired by IBMDOCL. On return to IBMDOCL the unused space left in the original 2K acquirement is freed.

If the length of the buffers to be acquired is unknown during compilation, buffer space is acquired by one of the transient modules. When this occurs an unused free area will normally be left between the transmitter and the buffer space. This is placed on the free area chain in the usual manner.

The open routines also alter the contents of certain fields in the FCB so that transmission statements will not result in a call to the open routines, as would occur in the case of an implicit open, described below.

Opening a File Implicitly

Implicit open is implemented by manipulation of the addresses to which transmission statements pass control so that these addresses always point to the open/close bootstrap if the file is not already open. This method is necessary as it is not always possible to determine during compilation which transmission statements will result in the opening of the file. (Implicit open is further explained under "Transmission Statements" below, and in figure 8.13.)

Transmission Statements

Compiled code calls the transmitter interface module IBMDRIO, passing to it the parameter list described above under the heading "Library-call record I/O statements" in the section "Compiler Output for Record I/O."

The interface module, IBMDRIO, first acquires a DSA, which is used both by IBMDRIO itself, and by the transmitter. It then initializes the registers and executes the instruction in the request control block (RCB). If the transmission statement being executed has been tested and found to be valid, the instruction will be a branch

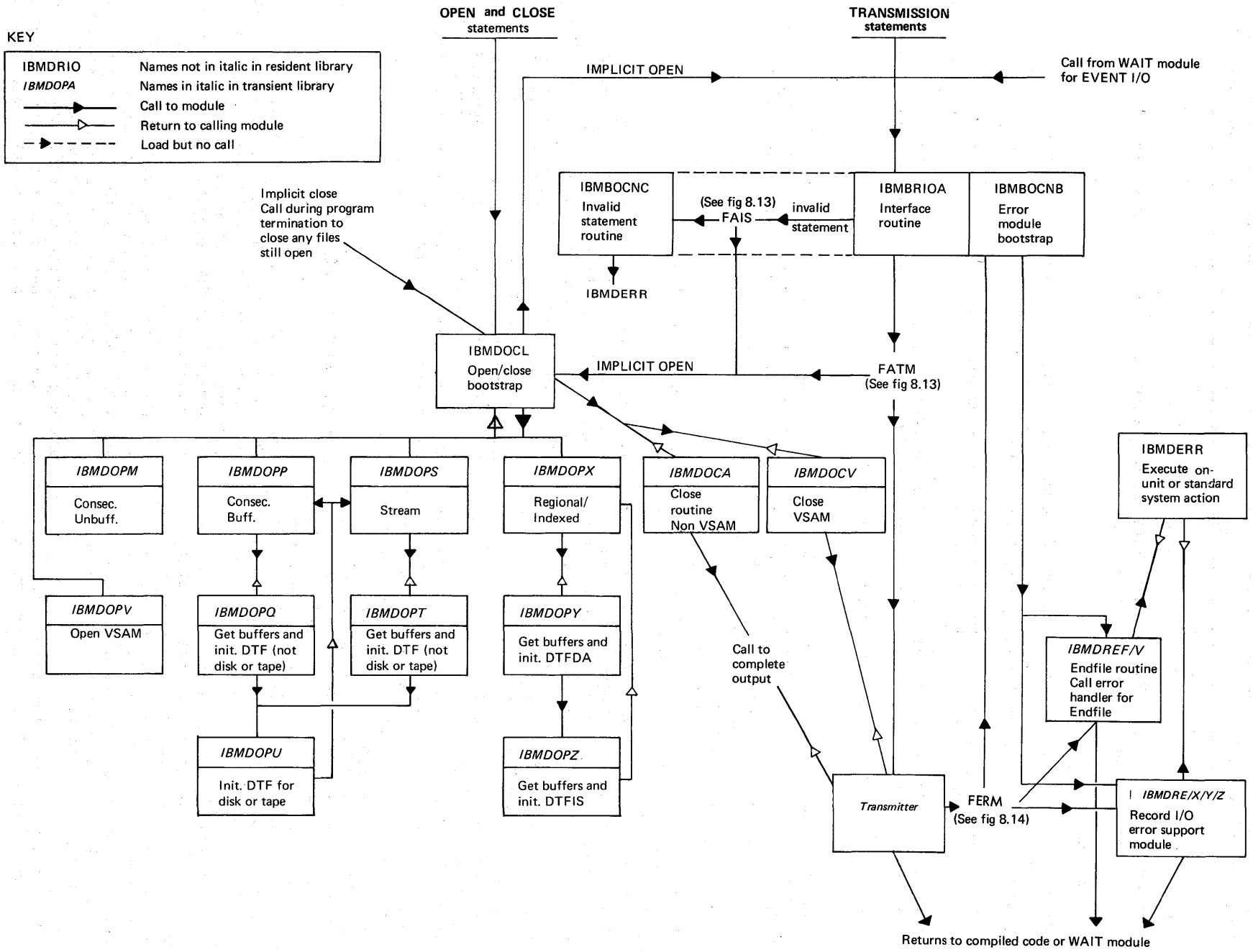


Figure 8.11. Organization of record I/O library modules

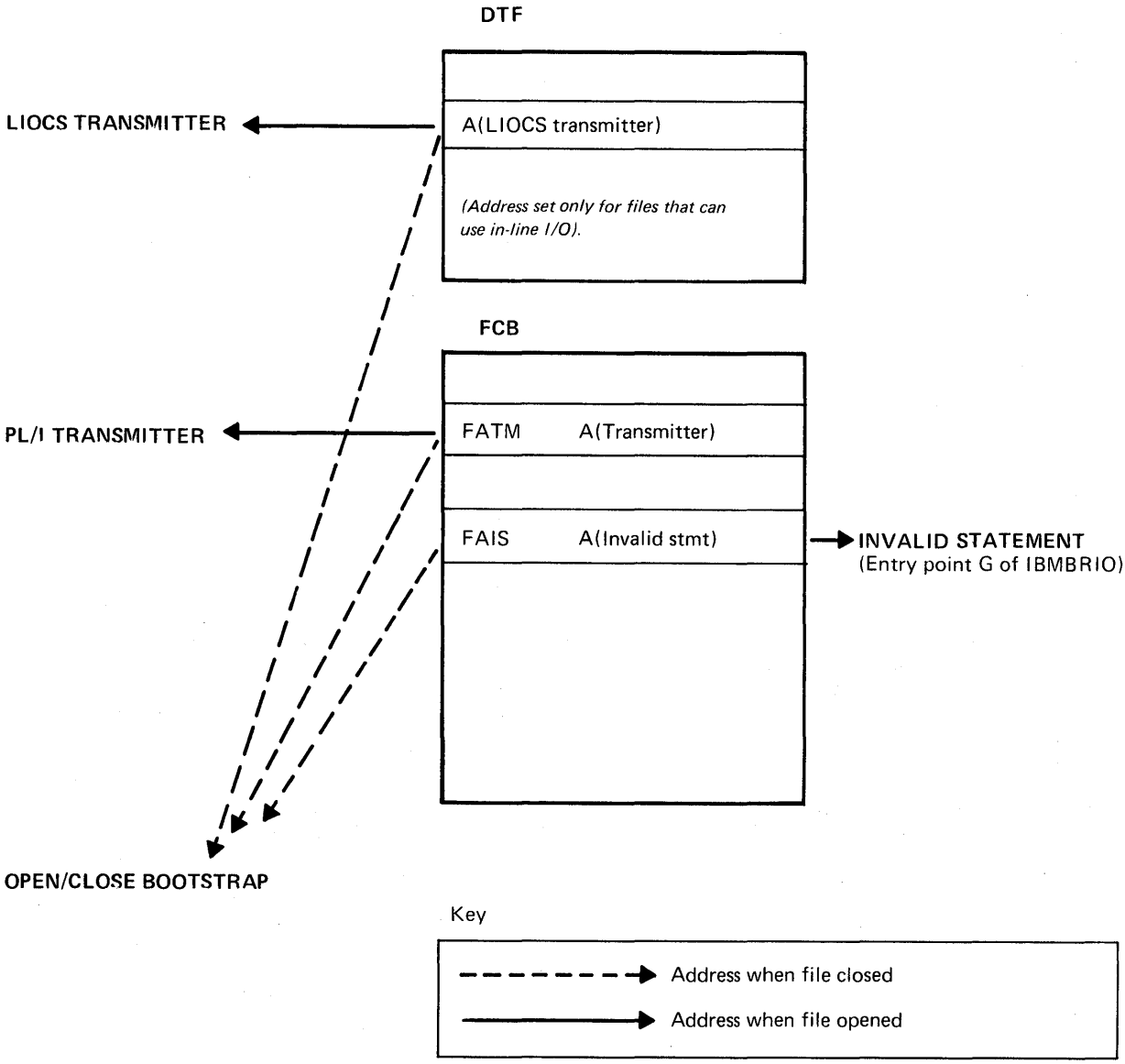


Figure 8.13. Implicit open procedure

instruction. If the statement has not been tested, or has been found to be invalid, the instruction will be a TM instruction. If the statement is valid, control will be passed directly, or after the TM instruction, to the address in the FCB field FATM. If the statement is invalid, control will be passed to another address, in the FCB field "FAIS."

As is shown in figure 8.13, the addresses held in these words depend on the condition of the file.

If the file is open, FATM contains the address of the PL/I transmitter, and FAIS contains the address of an entry point in the interface module, IBMBRIOC, which results in a call to the error handler.

Therefore, if the file is open, the transmitter will be called if the statement is valid; the error handler will be called if the statement is invalid.

If the file is not open, both FATM and FAIS contain the address of the open/close bootstrap routine IBMDOCL, entry point IBMBOCLC.

Therefore, if the file is not open, the execution of any transmission statement will result in a branch to the open/close bootstrap routine. The open/close

OPEN Statements	
1	Complete the DTF and the FCB for the file if necessary, or generate the ACB for VSAM files.
2	Obtain storage for buffers, index areas, etc. if necessary.
3	Issue the data management OPEN macro instruction.
4	Raise the UNDEFINEDFILE condition if the file cannot be opened.
Transmission statements	
1	Branch to the open routines if the file is not open.
2	Ensure that the statement is valid for the file.
3	Check the record and key variables for errors.
4	Issue the appropriate data management macro instruction.
5	Move the data between the buffer and the record variable if necessary.
6	Raise any conditions that occur during the execution of the statement.
CLOSE statements	
1	Ensure that all operations commenced on the file have been completed.
2	Issue data management close macro instruction.
3	Release any storage allocated to the file, e.g., I/O transmitter space.
Note:	The functions are not necessarily carried out in the above order.

Figure 8.12. Summary of work done by PL/I library routines

bootstrap routine calls the transient library open routines, which open the file, and alter the contents of FATM to point to the transmitter, and of FAIS to point to the error entry point IBMBRIOC. When the file is open, control is returned to the interface module and execution of the transmission statement is reattempted.

Implicit open for record I/O statements handled by in-line calls to data management are handled in a similar manner.

Transmitter Action

After the file is open and the statement validated, control is passed to the transmitter, which checks the record and key variables for errors, and issues the appropriate data management macro instruction. After data management has handled the request, control returns to the transmitter. The transmitter moves the data between the data management buffer and the record variable, or sets the pointer to the record, and checks to see whether any errors have occurred.

Transmitter modules do not acquire a DSA but use the DSA acquired by IBMDRIO.

For VSAM files a further control block, the IOCB (Input/Output Control Block) provided by OPEN, is used, in preference to the FCB, to hold information relating to the current statement. Space is provided in the IOCB for MODCB & SHOWCB parameter lists used respectively to modify and display the various fields in the data management RPL (Request Parameter List), which is passed to the VSAM LIOCS routine for any action macro. (See Appendix B for details of the IOCB). The transmitter action is essentially similar to that for other access methods. It is worth noting that a READ INTO will generally be implemented by means of a VSAM GET macro directly from the system buffer to the record variable. If the record variable is too short, VSAM will give a logical error return code and no transmission will take place, where upon the PL/I transmitter will reissue the request, providing an intermediate dummy buffer, and finally move the truncated record to the record variable.

Raising Conditions in Transmission Statements

To enable PL/I error handling to be available yet cause the mimimum possible overhead to error-free programs, transient-library modules are provided which are not loaded unless an error occurs. Two modules are available for all file types except VSAM file. VSAM files use one error module, IBMDREV and this handles both errors and end-of-file situations. The following discussion does not therefore apply to VSAM. The two types of error sorting used for non-VSAM files are:

1. The ENDFILE routine, IBMDREF, which can deal only with the ENDFILE condition.

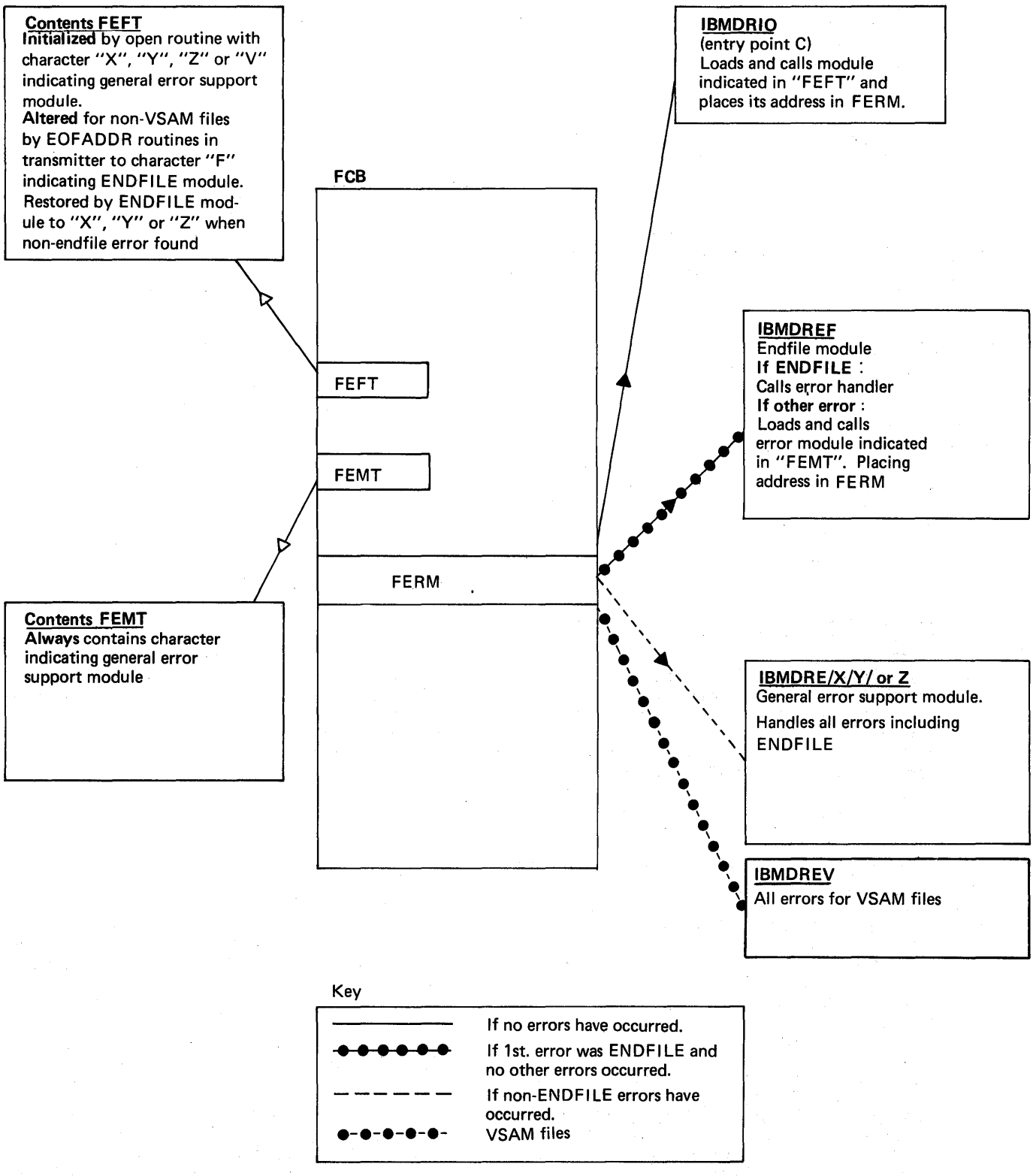


Figure 8.14. If conditions are raised during transmission, flow of control depends on the contents of the FCB field FERM

2. A general error module (one for each access method - see figure 8.10). This module is capable of handling all conditions that may arise, including ENDFILE, but is loaded only if the TRANSMIT, RECORD, KEY, or ERROR condition occurs

These transient error modules are all identified by the six letters IBMDRE followed by a further single character (see figure 8.10).

If a transmission error occurs, the transmission error routine within the transmitter will be entered, whether an in-line or library-call statement is being executed. Similarly, if end-of-file occurs, the end-of-file routine within the transmitter will be executed. Record and key errors are detected either by the transmitter or by compiled code.

When any of the errors or PL/I conditions mentioned above occurs during the execution of a record I/O statement, control is passed to the address held in the word "FERM" in the FCB. This address may be any one of the following:

1. The address of IBMDREF, the ENDFILE module.

2. The address of the general error module for the file type.

3. The address of a bootstrap routine, IBMBOCNB. This routine constructs the name of an error module by taking the skeleton IBMDRE*A and replacing the "*" by the letter in the single character field "FEFT" in the FCB. IBMDRIO then places the address of this module in FERM, and branches to the module it has loaded.

Thus by changing the contents of the field "FEFT", the transmitter can select a particular error module. The contents of "FEFT" is one of the following:

1. A character indicating the name of the general error module for the file type. This character is placed in "FEFT" during the execution of the OPEN statement.

2. The character "F", indicating the name of the ENDFILE module. The content of "FEFT" is changed to "F" by the end-of-file routine in the transmitter (which is entered when data management detects end-of-file).

Thus the module loaded by the bootstrap routine IBMBOCNB, and the address placed in "FERM", depends on whether end-of-file or another error is the first to occur on the file.

The result of this arrangement is that the general error module can be called in an end-of-file situation. Similarly, the ENDFILE module can be called when another type of error occurs, if ENDFILE was the first condition to occur.

To overcome this problem, the general error module contains code to handle ENDFILE, and the ENDFILE module contains code to test for other conditions, and load and call the general error module if appropriate.

The ENDFILE module restores the character in 'FEFT' from the field 'FEMT' and calls IBMBOCNB. FEMT always holds the character that identifies the general error module for the file. When the name has been constructed, the general module is loaded, its address is placed in FERM, and a branch is made to the module.

The process is illustrated in figure 8.14.

General Error Routines (Transient)

The general error routines set up a parameter list, and branch to the resident error-handler, IBMDERR, to handle the condition. If a normal return is made from an on-unit, the general error module will raise any further conditions that have occurred. After all conditions have been raised, a return is made to compiled code, or, in EVENT I/O, to the WAIT module.

ENDFILE Routine

The ENDFILE routine checks to ensure that the situation which has resulted in the call is really end-of-file, and, if so, passes control to the error handler.

CLOSE Statements

Files and data sets can be closed either by the PL/I CLOSE statement or by the termination of the program. In both cases, the close is carried out by the library routines. The bootstrap module IBMDOCL is called at entry point C or D, and it loads and calls the close routine, IBMDOCA. If any VSAM files are found, IBMDOCA loads and calls IBMDOCV.

The bootstrap routine is passed a parameter list containing the addresses of the FCBs for the files that require closing. IBMDOCA then closes these files. This may involve completing I/O operations and hence calling the transmitter. After handling any necessary transmission, IBMDOCA disassociates the file from the data set. If the transmitter was being used only by the file that is being closed, the storage for the transmitter is freed. A check is kept on the use of transmitter and other transient modules in a sixteen byte prefix at their head. The value held in this field is set to one when the transmitter is loaded, and decreased or increased as new files requiring the transmitter are opened. For implicit closing, the chain of open files starting in the TCA is scanned to determine which files must be closed.

When IBMDOCA has finished, it returns control (via IBMDOCL) either to compiled code (for an explicit close statement) or to the termination routine (for the end of the program).

If any VSAM files are found by IBMDOCA it makes a call to IBMDOCV for each VSAM file found. IBMDOCV then carries out similar action to IBMDOCA for the file. IBMDOCV is loaded for the first VSAM file-found. The space it occupies is freed by IBMDOCA before returning to IBMDOCL.

In-Line I/O Statements

Most transmission statements on buffered consecutive files are implemented by in-line calls to the LIOCS routines (see figure 8.2 for details). Such statements are referred to as "in-line I/O statements." Only READ, WRITE, and LOCATE statements are handled in this way. OPEN and CLOSE statements always result in library calls.

For in-line I/O, a call is made direct to the data management LIOCS routine whose address is held in the DTF. The DTF is addressed from the FCB. In addition to calling the LIOCS routine, compiled code moves the data as necessary to or from the record variable, or sets appropriate pointers. Compiled code may also check for the RECORD condition.

If there is an error in transmission, or if end-of-file is reached, the LIOCS routines will branch to the ERROPT or EOFADDR routines that are held in the PL/I transmitter. (The PL/I transmitter is always loaded by the open routines.) The ERROPT and EOFADDR routines set an error flag in the FCB and return to compiled code, normally via the LIOCS routine. If the error flag is on, or if the RECORD condition has occurred, compiled code branches to IBMBRIOD. This results in a call being made to the transient error module.

Typical code produced for an in-line I/O statement is shown in figure 8.6.

Control Blocks for In-Line Calls

For in-line I/O statements, the only control blocks that are set up are the FCB and DTF. The request control block, and record and key descriptor are not required as the information is known during compilation and suitable code to move the data to or from the record variable can be generated. The RCB is not generated, as it is required only by the library routines to determine the statement type when checking statement validity.

Implicit Open for In-Line Calls

Implicit open for in-line calls is handled in a similar way to that used for library calls (described above).

For a compiled code call, the address in the DTF that normally holds the address of the data management LIOCS transmitter is initialized to point to the open/close bootstrap routine, IBMDOCL (see figure 8.13). When the open routines have finished, the address in the DTF is altered to point to the LIOCS routine.

If the file is successfully opened, a test is made to see whether the entry to IBMDOCL was for an in-line call and, if it was, control is passed to the data management address held in the DTF. For input, this causes the LIOCS transmitter to be entered and a return made to compiled code. For output, it will cause entry into code in the PL/I library transmitter which places the address of the LIOCS transmitter in the DTF and returns to compiled code. This is the normal transmitter action for the first output statement.

Event I/O

Event I/O is fully described in chapter 11, under the heading "The WAIT Statement." The principles are described briefly below.

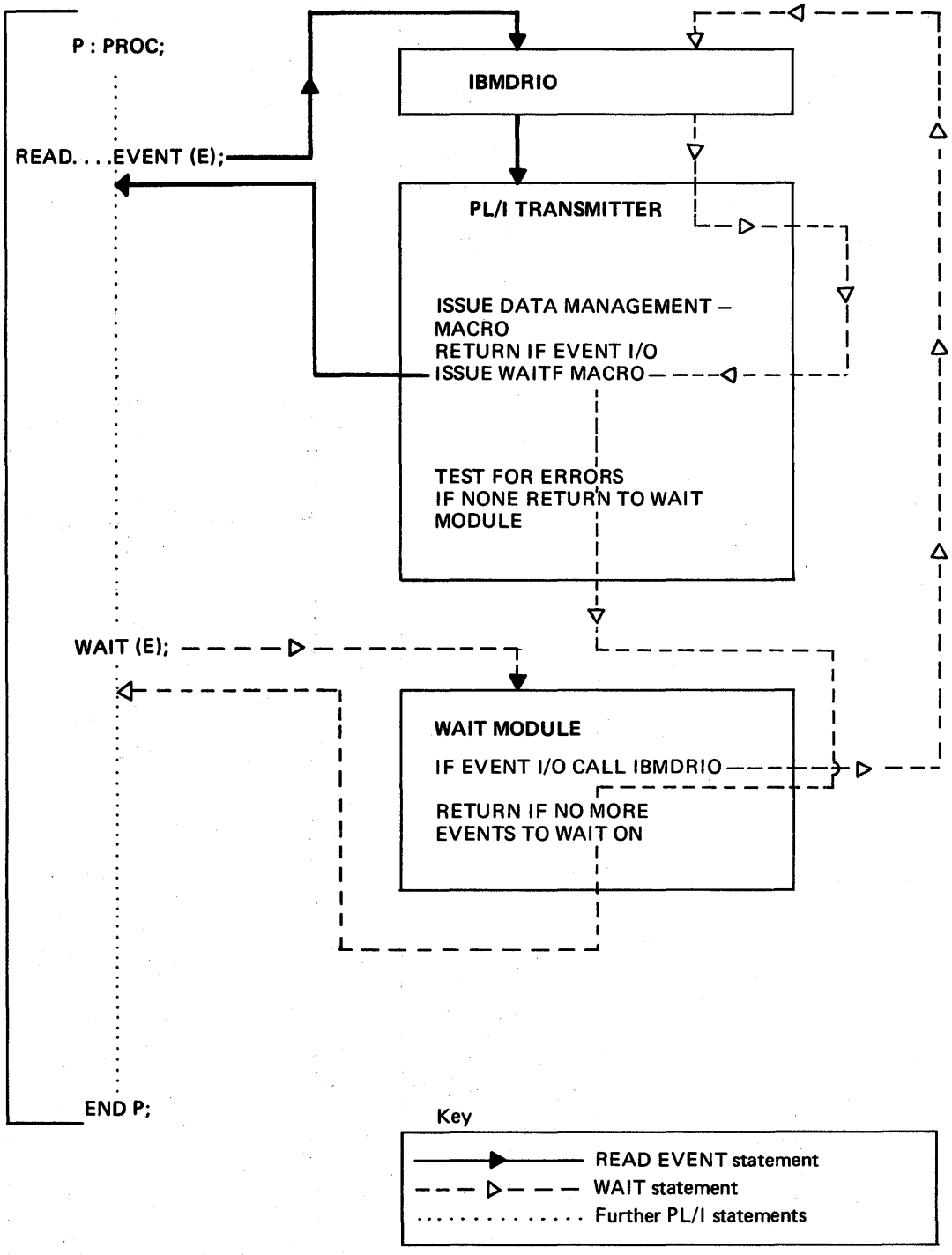


Figure 8.15. Flow of control for READ, EVENT and WAIT statements

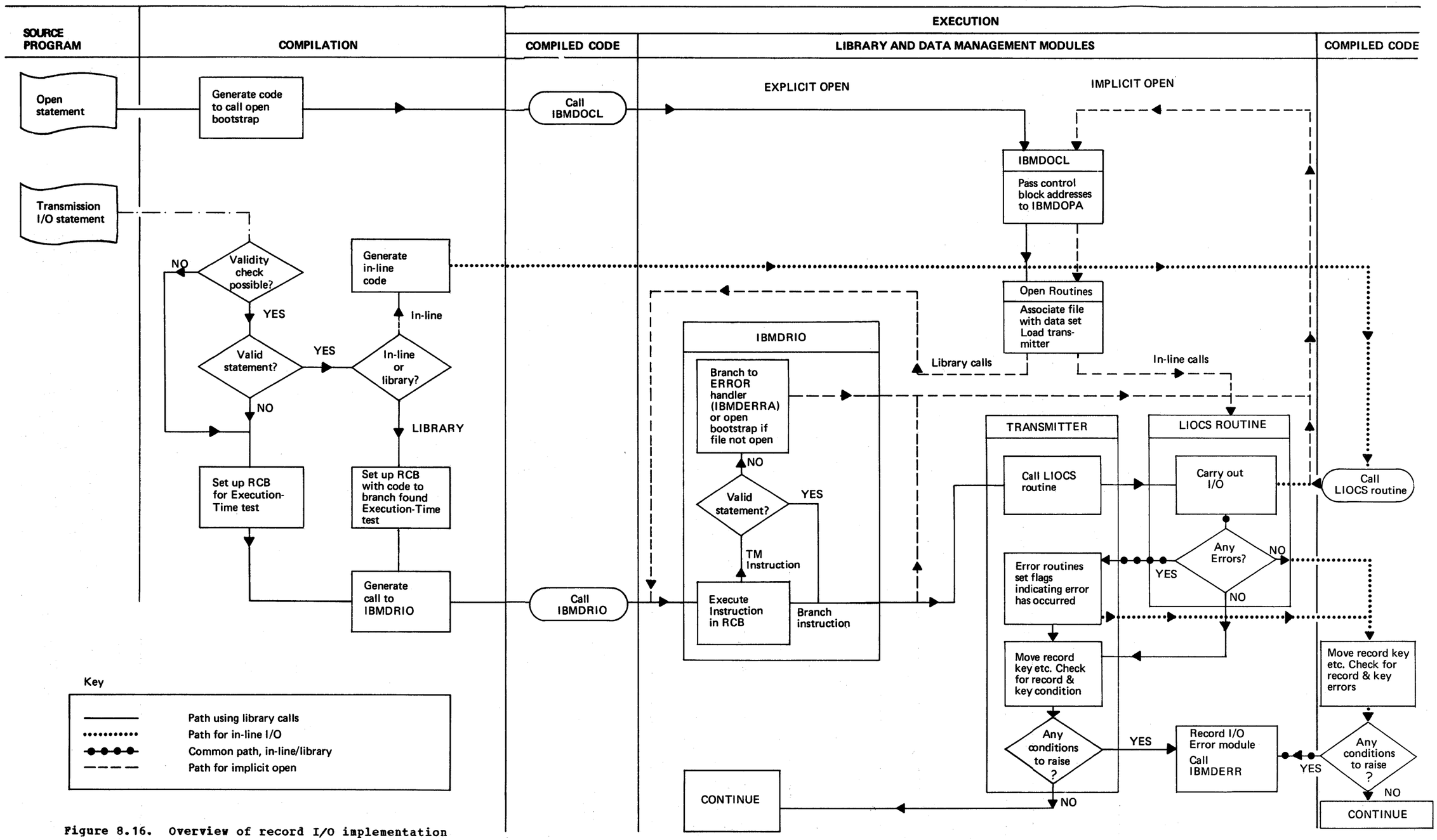


Figure 8.16. Overview of record I/O implementation

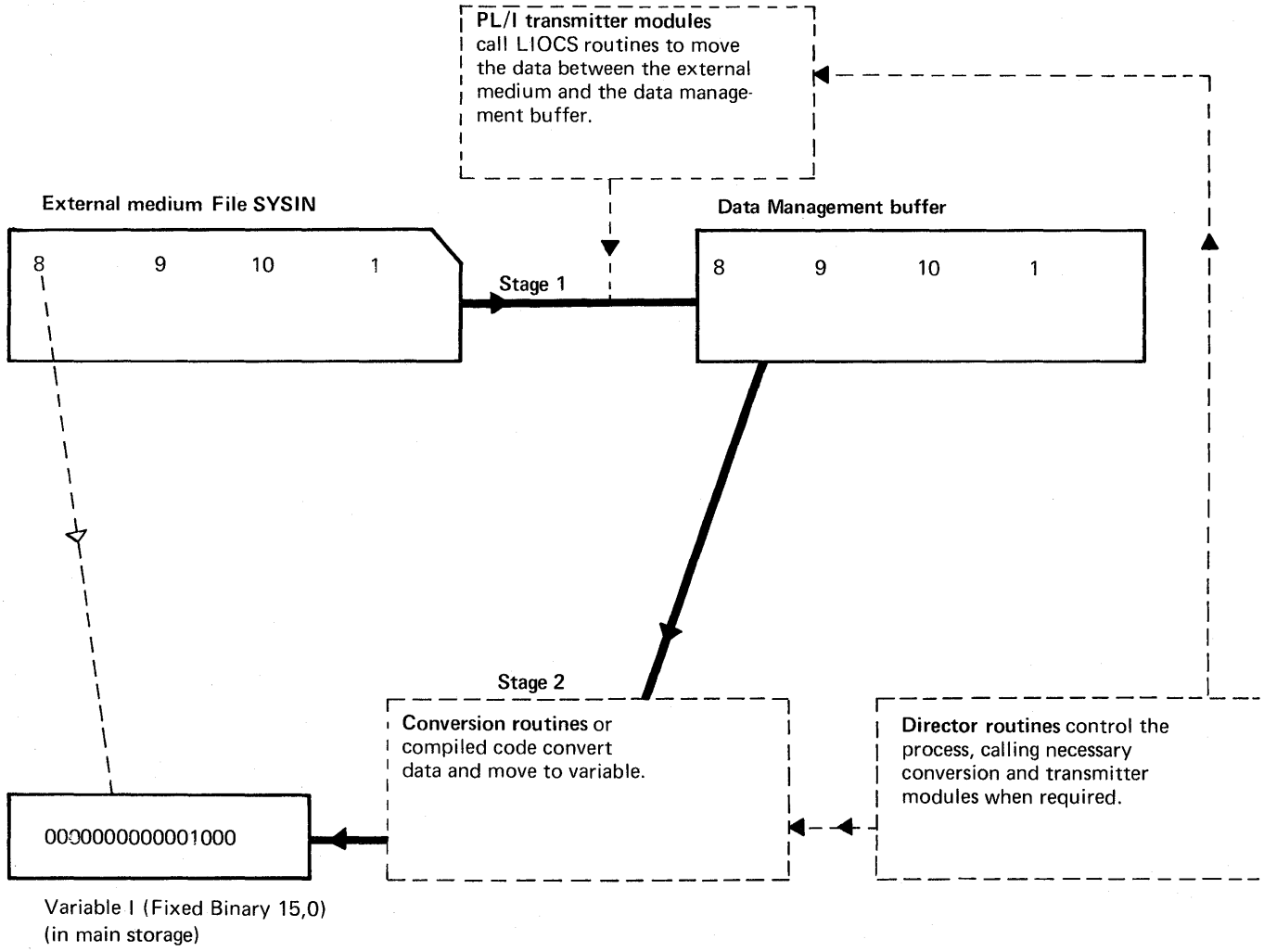
If the EVENT option is used in an I/O statement, the statement will be handled by a library call. Thus control will be passed from IBMDRIO to a PL/I library transmitter. This transmitter returns control to compiled code as soon as the data management macro instruction is issued. When the WAIT statement nominating the event is reached by compiled code, a call is made to the WAIT module which returns control to the transmitter via IBMDRIO.

A WAITF macro instruction is then issued, and control is returned to the transmitter when the input or output operation is complete. The transmitter then tests for errors and, providing no errors have been detected, returns control to the WAIT module, which returns control to the next statement.

For VSAM files EVENT I/O is simulated. The return code from data management is tested by the transmitter immediately after issuing an action macro instruction. However, any errors detected are held over until the corresponding WAIT statement is issued when control is returned to the transmitter.

Figure 8.15 illustrates the principles used in EVENT I/O.

PL/I Statement: GET LIST(I);



Stream input/output is a two stage process. The data is moved between the external medium and the data management buffer, and between the buffer and the variable. Any necessary conversions are made between the buffer and the variable. The operation is controlled by director modules. The director modules call the appropriate routines to do the transmission and conversion. Transmission is carried out in a similar way to that used for RECORD I/O.

Note that a further input statement will require the value 9 which is already in the data management buffer. *Consequently the transmitter need not be called and a pointer must be kept to the position reached in the buffer.*

Figure 9.1. Conceptual diagram of stream I/O

Chapter 9: Stream-Oriented Input/Output

Note on Terminology

In this chapter, the terms source and target are used when referring to transfer of data. The source is the point from which the data is taken; the target is the point to which it is moved, possibly in a converted format.

Introduction

PL/I stream-oriented input/output allows the programmer to move data between a PL/I variable and an external medium without any concern about internal and external data types or any attention to record boundaries; both conversion and record boundary problems are handled automatically.

Although it appears to the programmer that the data is moved directly between the external medium and the PL/I variable, the move is in fact a two stage process, as shown in figure 9.1. In the first stage, the data is moved to a data management buffer, in the second stage, it is moved from the buffer to the target. When the data is moved to or from an external medium, a complete record is always moved. When the data is moved to or from a PL/I variable only as much data as is contained in the variable is moved. The amount of data moved in the one stage need bear no relation to the amount moved in the other. Thus synchronization of the two stages is the principal job in implementing stream I/O.

Transmission between the buffer and the external medium is handled by the LIOCS routines of data management. These routines are called by PL/I transient library transmitters in a way similar to that used in library call record I/O. The movement between the buffer and the PL/I variables is generally handled by the PL/I conversion routines.

Data items transmitted by stream I/O are not affected by record boundaries (see figure 9.2). There may be any number of data items in a record, and an item may span any number of records. Because the LIOCS routines make only one record available to the program at any one time, a method is needed to build up complete items if they span the record boundary. Similarly, because GET and PUT statements may read or write less than a complete record, a method is needed of keeping track of the position reached in the record, so that the next GET or PUT can start from the correct position.

Operations in a Stream I/O Statement

A stream I/O operation can involve any or all of the following opertions:

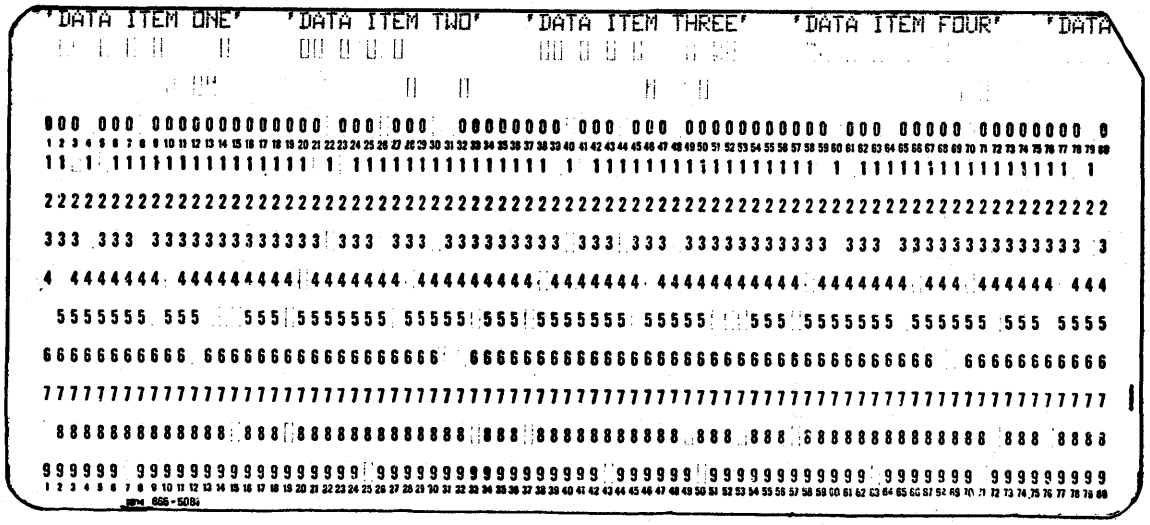
1. Opening the file, and raising the ERROR condition if the statement is invalid.
2. Keeping track of the position in the buffer.
3. Calling the transmitter for a new record.
4. Building in intermediate workspace an item too large to be held in the current record.
5. Determining which conversion is required, and calling the routine to carry out the conversion.

Control of operations (2) through (5) is handled by director routines. For list-directed and data-directed I/O, PL/I library director routines are used. For edit-directed I/O, the job is shared between library routines, compiler-generated subroutines, and compiled code.

Before the director module or director code receives control an initialization module is called. This module handles item (1) in the list above: checking statement validity, and opening the file if it is not already open.

Because there are three modes of stream I/O, the exact situation cannot be defined in a generalized discussion or diagram. However, the basic principles are shown in figure 9.3. The sequence is:

1. A call to the initialization module.
2. A return to compiled code.
3. A call to the director module.



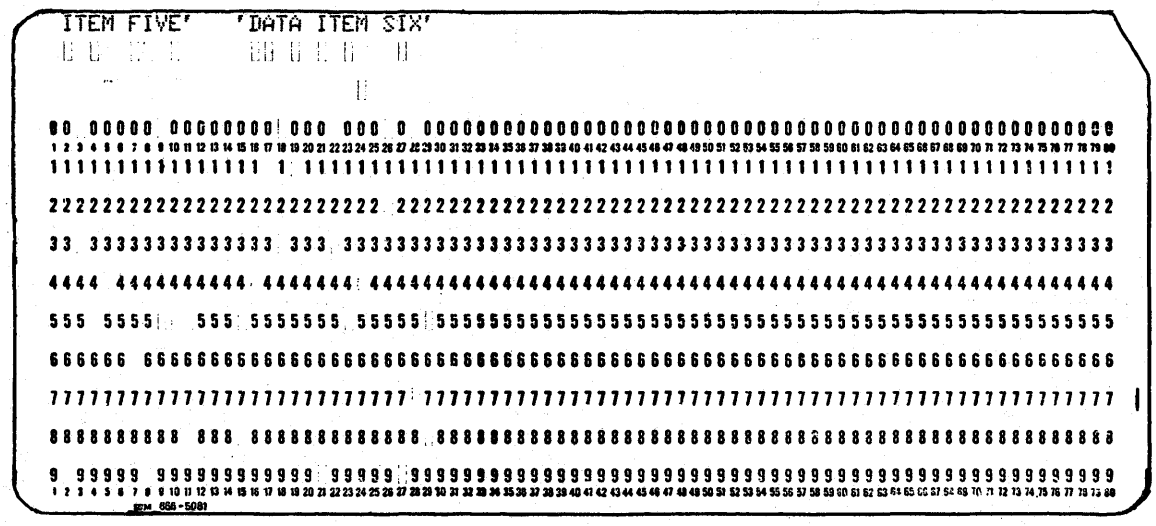


Figure 9.2. Record boundaries do not affect stream I/O

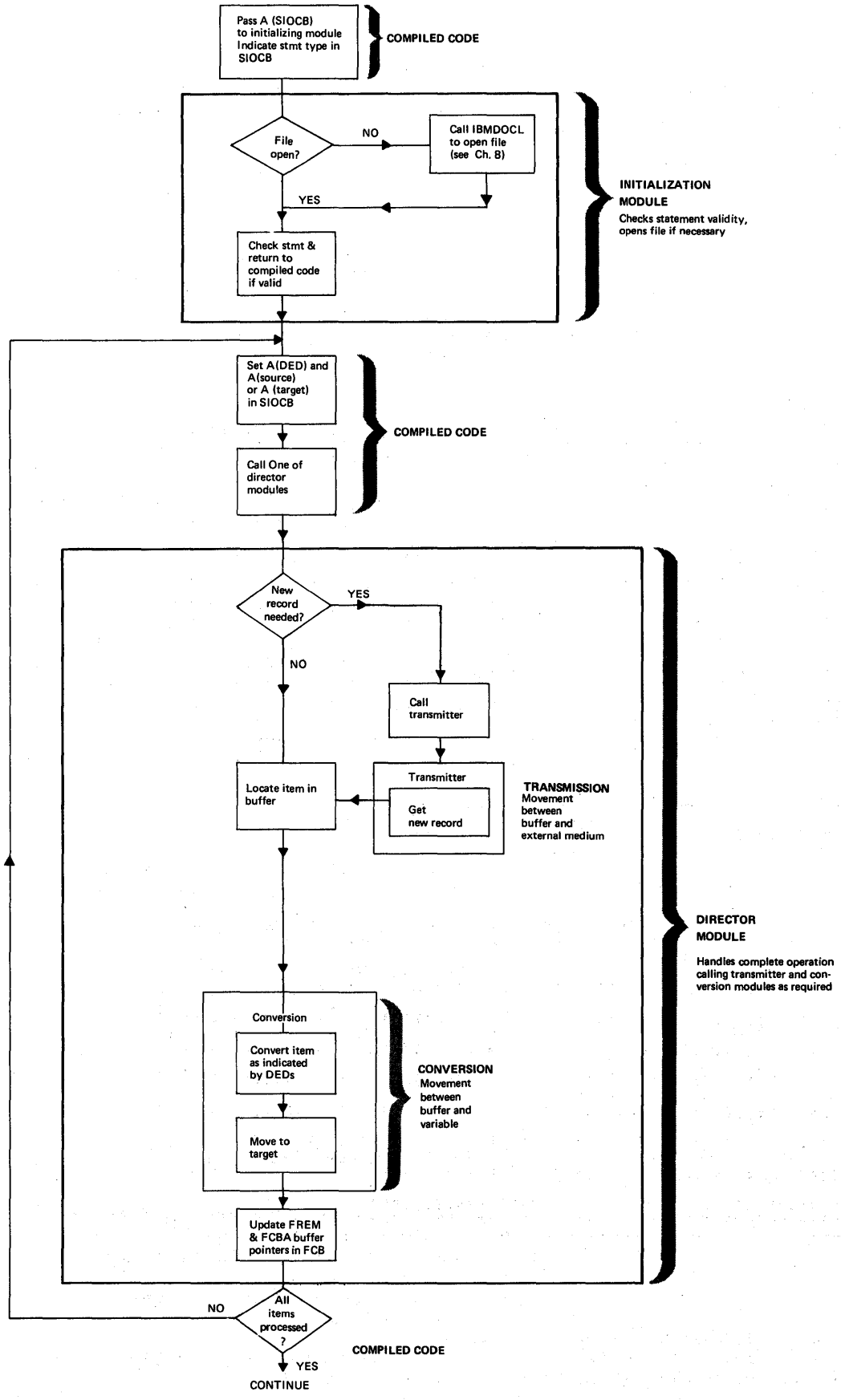


Figure 9.3. Generalized flowchart of a stream input statement

Stream I/O Control Block (SIOCB)

To simplify communication between the large number of routines that may be used in a stream I/O operation, a control block is set up for the duration of execution of the stream I/O statement. This control block is known as the stream I/O control block (SIOCB). The contents of the SIOCB are shown in figure 9.4.

SSRC	Address of source or source locator
SSDD	Address of source DED
STRG	Address of target or target locator
STDD	Address of target DED
SFLG	Flag bytes
SFCB	Address of FCB for file
SRTN	Abnormal return address (next statement)
SAVE	Save word used by compiler
SCNT	Count of items transmitted (Halfword)
SOCA	Address of ONCA
SSTR	Area present only for GET or PUT STRING, to hold a dummy file block.

Figure 9.4. Stream I/O control block (SIOCB)

Basically, the SIOCB consists of the addresses of the source and target (or their locators), of the DEDs of the source and the target, and of various other items. The SIOCB can be passed directly to the conversion modules, because the first four words are the same as the parameters expected by the conversion routines.

File Handling

In stream I/O, file organization is always sequential and the access method used is the queued sequential access method (QSAM).

Transmission

Transmitters are called by the director modules or by the close module to complete transmission when the program is terminated.

As with record I/O, LIOCS transmitters are used, and they are called by PL/I transmitters. The PL/I transmitters contain the ERROPT and EOFADDR routines, which are entered when end-of-file or other errors are detected in the LIOCS routines. Seven different transmitter modules are used in stream I/O; they are listed in the summary of subroutines at the end of this chapter.

Opening the File

The same basic method is used for opening the file as is used for record I/O. During compilation, a define-the-file control block (DTF), a file control block (FCB), and an environment control block (ENVB) are set up. At open time, the information addressed from the ENVB is used to complete the FCB and DTF, the PL/I transmitter is loaded, and its address is placed in the FCB. The LIOCS routine to be used is determined during compilation, and link-edited.

Implicit Open

Implicit opening is handled by the initialization routines, which check to see whether the file is open and, if not, call the open/close bootstrap routine IBMDOCL.

The FCB for stream I/O is similar to that used for record I/O. However, it contains certain additional fields which are needed only for stream I/O. The most important of these fields are the buffer control fields.

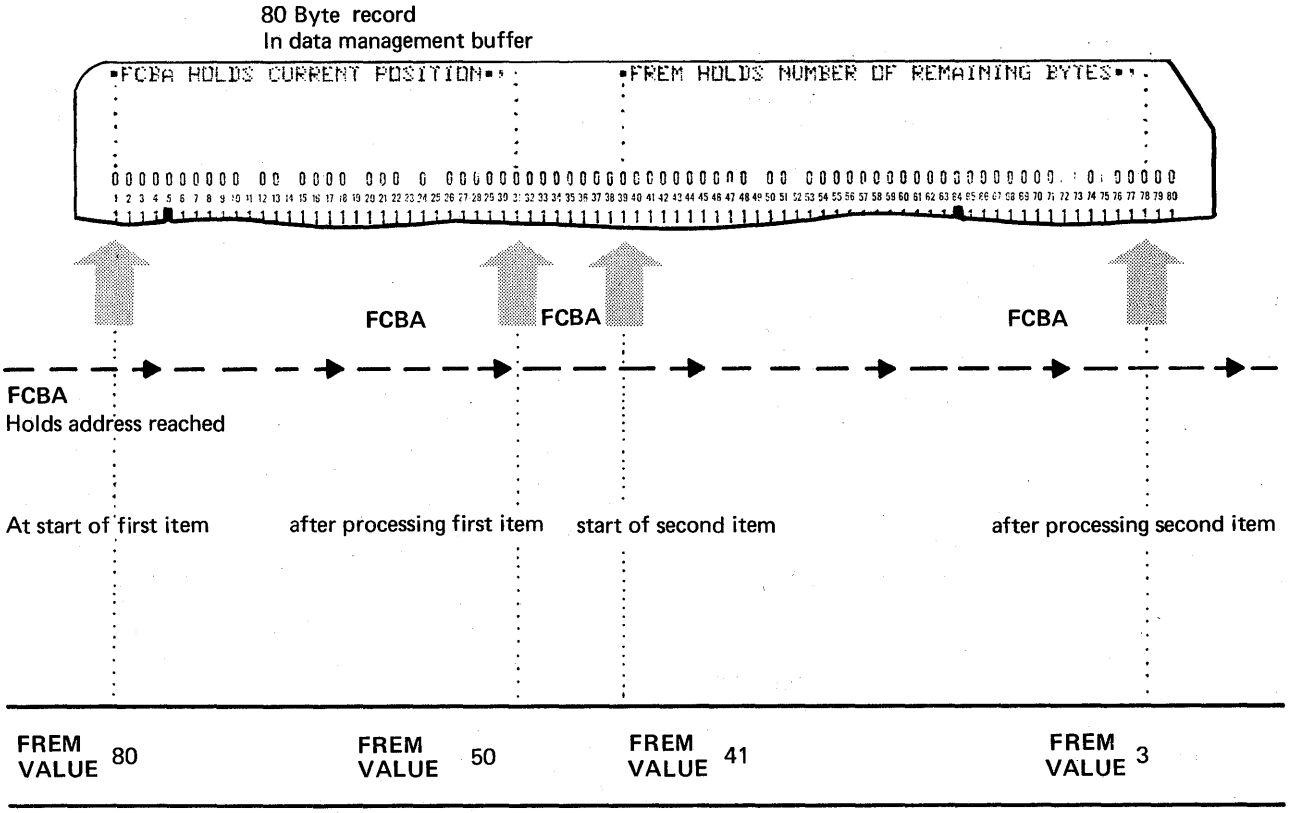
Keeping Track of Buffer Position

Two fields in the FCB are used to keep track of the position which has been reached in the data management buffer, and to indicate when a new record will be required. These fields are the buffer control fields:

1. FCBA - pointer for position reached in current record.

PL/I STATEMENT:

```
GET FILE (SYSIN) LIST (A, B);
```



FREM holds number of remaining bytes

Figure 9.5. The FCBA and FREM fields of the FCB

2. FREM - number of unused bytes remaining in the record.

FCBA points at the position reached in the record and enables the director routines to identify where the next input item must be read from, or where the next output item must be written. FREM contains the number of bytes left in a record. It enables the director modules to determine when a new record will be required, and whether an item is too large to be held in the remainder of the record and will consequently require intermediate workspace. Figure 9.5 illustrates the use of FCBA and FREM.

Handling the Conversions

Conversions in stream I/O are normally handled by the library conversion package. The conversion package, described in chapter 10, consists of conversion routines and conversion director routines. Conversion director routines examine the DEDs of the source and the target passed in the argument list (i.e., the SIOCB, for stream I/O), and determine which entry point of which conversion module is required. Each conversion has a unique entry point.

A number of conversion director modules are used exclusively by edit-directed stream I/O. These are called external

conversion directors, and are listed in the summary of subroutines at the end of this chapter. Each module corresponds to a particular format of input/output. When the type of input or output has been determined by the director modules, the appropriate conversion director routine can be called to handle the conversion.

In edit-directed I/O, the conversion required is normally predictable during compilation, because it is implied in the format list. Consequently, the conversion modules can be called from compiled code rather than from the stream I/O director routines. Alternatively, compiled code may handle the conversion in-line.

When a library conversion module is required by compiled code, the conversion director module may be called, or the conversion module itself may be called. When the conversion module is called, compiled code must carry out the jobs normally handled by conversion director modules, that is, setting up a number of fields that are used mainly in handling CONVERSION and other PL/I conditions.

Handling GET and PUT Statements

There are considerable differences in detail between the handling of GET and PUT statements for the three different modes of stream I/O. However, they all follow the basic scheme in figure 9.3 and summarized above under the heading "Operations in a Stream I/O Statement."

The implementation of GET and PUT statements is covered in some detail below for list-directed I/O. For data-directed and edit-directed I/O, the differences from list-directed are highlighted.

List-Directed GET and PUT Statements

PUT LIST Statement

Implementation of a list-directed output statement is shown in figure 9.6. The process consists of four steps:

1. Compiled code calls the initialization routine, passing the address of the FCB and of the SIOCB, in which compiled code has set flags indicating the statement type.

2. The initialization routine, IBMDSIO, calls the open routine if the file is not open, and checks the validity of the statement. If the statement is invalid, a branch is made to the error handler, passing an error code indicating "invalid statement." This results in a message being generated, and the ERROR condition being raised. If the statement is valid, control is returned to compiled code.

   IBMDSIO also handles any format options, by calling the formatting module IBMDSPL. Control then returns to compiled code.

3. Compiled code places the address of the source (or its locator, if the item is a string) and the address of the source DED in the SIOCB. (See chapter 4 for information on locators.) Compiled code then calls the director module.

4. The director module completes the SIOCB with the address of the target locator and the address of the DED of the target. The target locator gives the length required for the item. As the target is always a character string, a locator will always be used for it. The address of the target is a position in the buffer. For PRINT files, the position is indicated in the tab table, which will either have been set up by the programmer by use of PLITABS, or be the default tab table in the library module IBMBSTA. For non-print files, a one-byte space follows each item. When the starting postion for the item has been determined, the director module determines whether there is enough space in the output buffer for the converted item. If there is not, the director determines whether this is because there is no room left in the buffer, in which case it is simply a new record that is needed, or because there is insufficient room in one buffer, in which case the item will have to span a record boundary.

   If it is simply a case of acquiring a new record, the director calls the transmitter to acquire it. The director then calls the appropriate conversion routine, passing it the SIOCB as a parameter list. The conversion routine will then move the data from the PL/I variable to the new record in the data management buffer.

   If, however, the converted item will span the boundary between the current and subsequent records, intermediate workspace is acquired in the form of a VDA (variable data area). The converted item is then placed in the

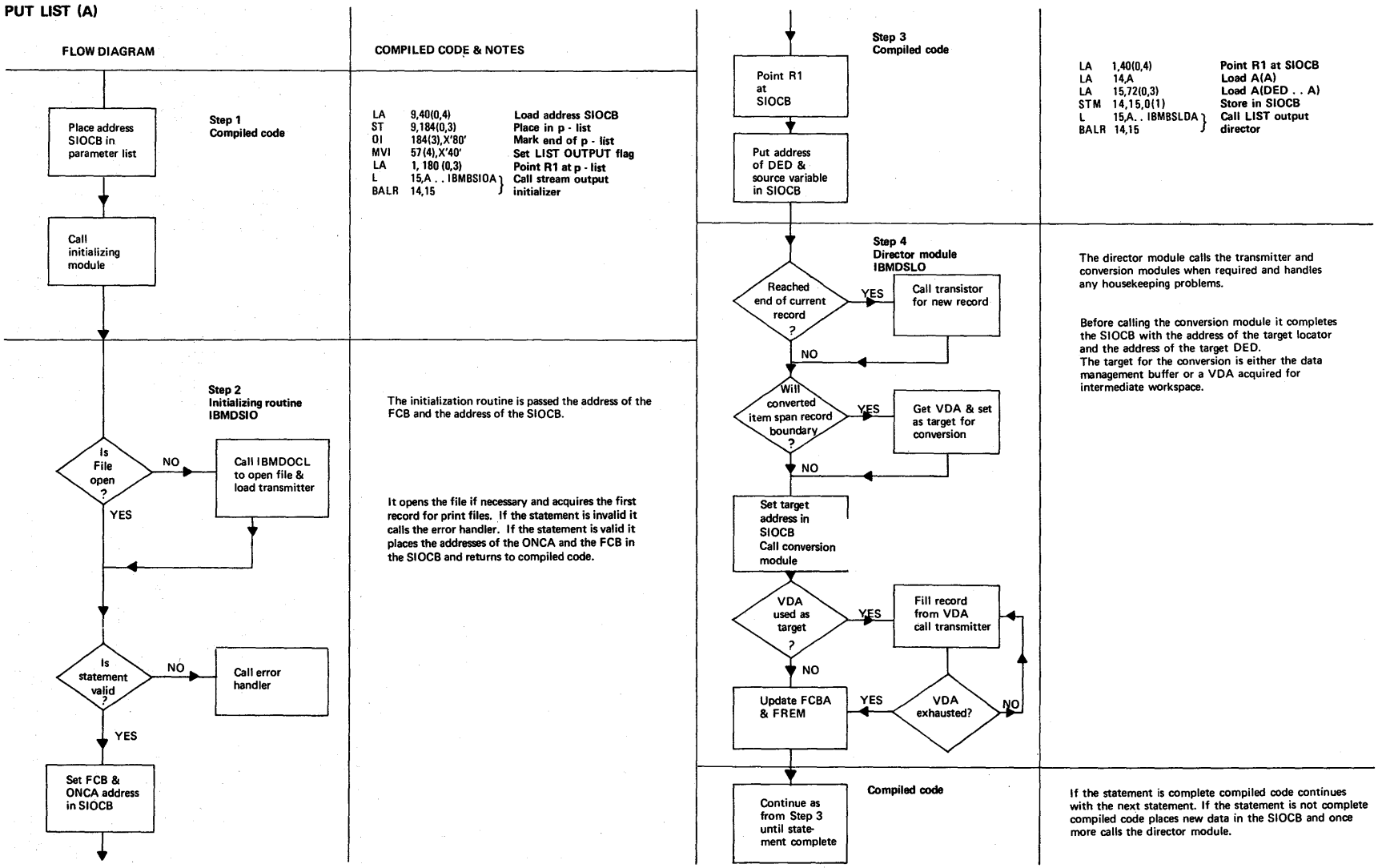


Figure 9.6. List-directed output statement

```
* STATEMENT NUMBER  2
00005E   41 90 D 0C8     LA      9,200(0,13)       Load address of SIOCB
000062   50 90 3 044     ST      9,68(0,3)         Store in parameter list
000066   96 80 3 044     OI      68(3),X'80'       Set end of parameter list flag
00006A   92 40 D 0D9     MVI     217(13),X'40'     Set flags in SIOCB
00006E   41 10 3 040     LA      1,64(0,3)         Point R1 at parameter list
000072   58 F0 3 024     L       15,A..IBMBSIOA    Branch and link to initialization routine
000076   05 EF           BALR    14,15
000078   41 E0 D 0A8     LA      14,A              Load address of A
00007C   41 F0 3 030     LA      15,DED..A         Load address of DED..A
000080   41 10 D 0C8     LA      1,200(0,13)       Load address of SIOCB
000084   50 10 D 0C0     ST      1,192(0,13)       Store in temporary storage
000088   90 EF 1 000     STM     14,15,0(1)        Place address of A and DED..A in SIOCB
00008C   58 F0 3 028     L       15,A..IBMBSLOA    Branch and link to list directed director routine
000090   05 EF           BALR    14,15
000092   41 E0 D 0AC     LA      14,B              Load address of B
000096   58 10 D 0C0     L       1,192(0,13)       Point R1 at SIOCB
00009A   50 E0 1 000     ST      14,0(0,1)         Place address of B in SIOCB
00009E   58 F0 3 028     L       15,A..IBMBSLOA    Branch and link to list-directed director routine
0000A2   05 EF           BALR    14,15
```

Figure 9.7. Typical code generated for a PUT LIST statement

VDA. As much of the data as will fit is moved into the data management buffer, and a new record is acquired by a call to the output transmitter. The new record is then filled. This process is continued until the complete item has been moved into buffers. Finally, FCBA and FREM are updated.

If there are further data items to be handled, a return is made to step (2), and the address of a new source field and its DED are placed in the SIOCB. This process is continued until all items in the data list have been processed. The object code produced for a PUT LIST statement is shown in figure 9.7.

GET LIST Statement

GET LIST statements follow the same sequence, but the transmission is in the opposite direction. The main differences are:

1. If record spanning is involved, the item is assembled in intermediate workspace before it is converted.

2. A locator is built for the source string from the input, and the addresses of the locator and of the DED for the source are placed in the SIOCB by the director module. For input, the address of the target or its locator and the address of the target DED are placed in the SIOCB by compiled code.

3. FCBA and FREM are updated before the item is converted.

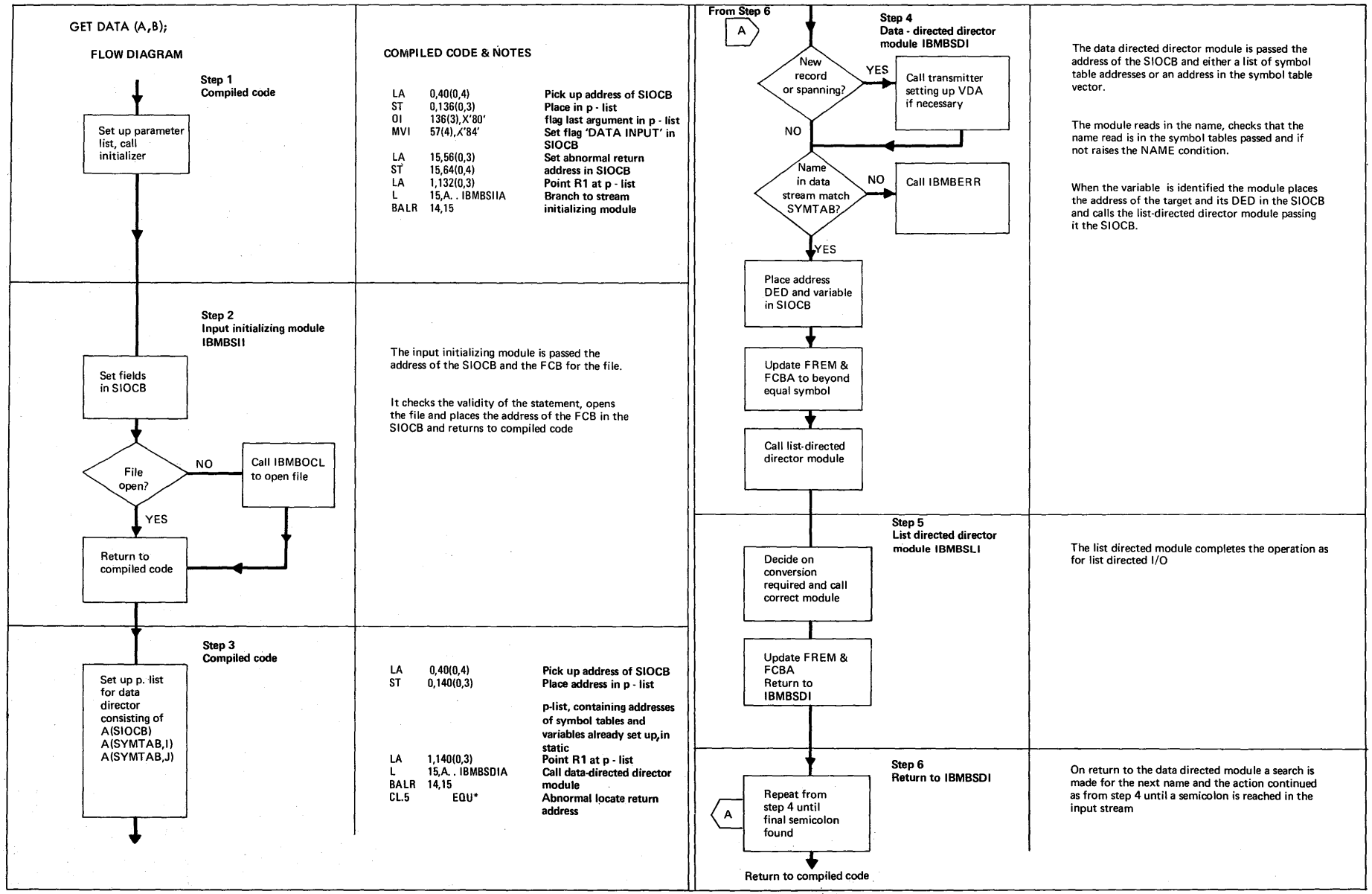


Figure 9.8. Data directed input statement

```
* STATEMENT NUMBER  3
0000A4  41 90 D 0C8          LA    9,200(0,13)      Load address of SIOCB
0000A8  50 90 3 044          ST    9,68(0,3)        Store in parameter list
0000AC  96 80 3 044          OI    68(3),X'80'      Mark end of parameter list
0000B0  92 80 D 0D9          MVI   217(13),X'80'    Set flags in SIOCB
0000B4  92 01 D 0DA          MVI   218(13),X'01'
0000B8  41 10 3 040          LA    1,64(0,3)        Point R1 at parameter list
0000BC  58 F0 3 024          L     15,A..IBMBSIOA
0000C0  05 EF                BALR  14,15            Branch to initialization routine
0000C2  41 90 D 0C8          LA    9,200(0,13)      Load address of SIOCB
0000C6  50 90 3 048          ST    9,72(0,3)        Store in parameter list
0000CA  96 80 D 0DB          OI    219(13),X'80'    Set flag in SIOCB
0000CE  41 10 3 048          LA    1,72(0,3)        Point R1 at parameter list
0000D2  58 F0 3 020          L     15,A..IBMBSDOA
0000D6  05 EF                BALR  14,15            Branch to data-directed director routine
```

Figure 9.9. Typical code generated for a PUT DATA statement

Data-Directed GET and PUT Statements

Data-directed GET and PUT statements follow a similar sequence to list-directed statements, in that there is first a call to the initialization module, followed by a call to a director routine. However, the data-directed director module is passed a means of identifying the names and addresses of all the variables involved in the statement rather than one item at a time.

When the data-directed module has identified the location of the variable to or from which the data is to be moved, it calls the list-directed director module which then handles the movement of the value of the variable. When the value of the variable has been transmitted, control returns to the data-directed module. The data-directed director then handles the next name, determines the address of the variable associated with the name, and calls the list-directed director module to handle the transmission of the value. This process continues until the statement is complete. The process is illustrated in figure 9.8.

The list-directed director module is called separately for each item. It is passed the SIOCB with the addresses of the source or target (or its locator) and the address of its DED correctly set up by the data-directed director module. The item is then handled as if it were a list-directed item.

If a data list is included in the statement, the source or target variables are identified from a list of symbol tables. If no data list is included in the statement, they are identified from the the symbol table vector.

A symbol table associates a name with

the address of a variable. The symbol table vector for an external procedure is a list of the symbol tables known in the external procedure. The list is arranged in program block order. When a symbol table vector is used, the address passed is the start of entries for items known in the current block. Symbol tables and the symbol table vector are described further in chapter 4. Their format is shown in appendix B.

The object code produced for a PUT DATA statement is shown in figure 9.9.

Edit-Directed GET and PUT Statements

Edit-directed I/O differs from the other modes of stream I/O in that the conversions required and the positions in the record where an item is to be placed or will be found are indicated in the format list of the I/O statement.

The format list contains two related types of information:

1. The type and length of the item (e.g., F(3), A(25), etc.), known as data format information.
2. Spacing information (e.g., X(3),COL(70),etc.), known as control format information.

Both types of information are compiled as format DEDs (FEDs) and are passed by compiled code to the routines that require the information.

Because the information is available during compilation, it is normally possible for the compiler to determine the conversions that will be required. Consequently compiled code can call the required conversion or conversion director routine, or generate in-line conversion code without the assistance of a library director module.

Compiler-Generated Subroutines

To further optimize edit-directed I/O, a number of compiler-generated subroutines have been provided. They carry out the following functions:

1. Keeping track of the buffer position, freeing and acquiring intermediate workspace where necessary, and calling the library when a new record is required.
2. Handling X format control items, except where a new record is required.

These compiler-generated subroutines have the advantage over library modules that they are not external, and consequently do not have to follow the external calling conventions.

The compiler-generated subroutines are supported by two types of library director module:

1. Two short modules, IBMDSEO and IBMDSEI, that interface with the transmitter and are called by the compiler-generated subroutines when a new record is required.
2. Three routines that handle the complete processing of an item (as does the director for list-directed I/O). These routines are called when an item cannot be handled by compiler-generated subroutines.

   IBMDSED is used only when complex data or format items appear in the program.

   IBMDSEE is used when both GET and PUT EDIT statements are used in the program and no complex data or C-format items appear. This routine contains the functions of the formatting module IBMDSXC and the conversion director modules IBMBSFI,IBMBSFO, and IBMBSAO, and the A-format function of IBMBSAI. Consequently, it normally uses less space than IBMDSED which calls these modules.

   IBMDSEH is used when only edit-directed output is used in a program and no complex data is used. IBMDSEH is similar to IBMDSEE but does not contain the input code.

The superset/subset feature of the linkage editor ensures that only one of the modules is link-edited if ESD references are made to IBMDSEE and IBMDSEH. IBMDSED can handle all situations, and IBMDSEE can handle any situation handled by IBMDSEH.

The decision on whether to use compiler-generated subroutines or the overall library director module is made at compile time. Figure 9.10 shows the conditions under which each method is used.

A typical edit-directed statement takes the form:

1. A call to the initialization module to open the file (if necessary), and check statement validity.

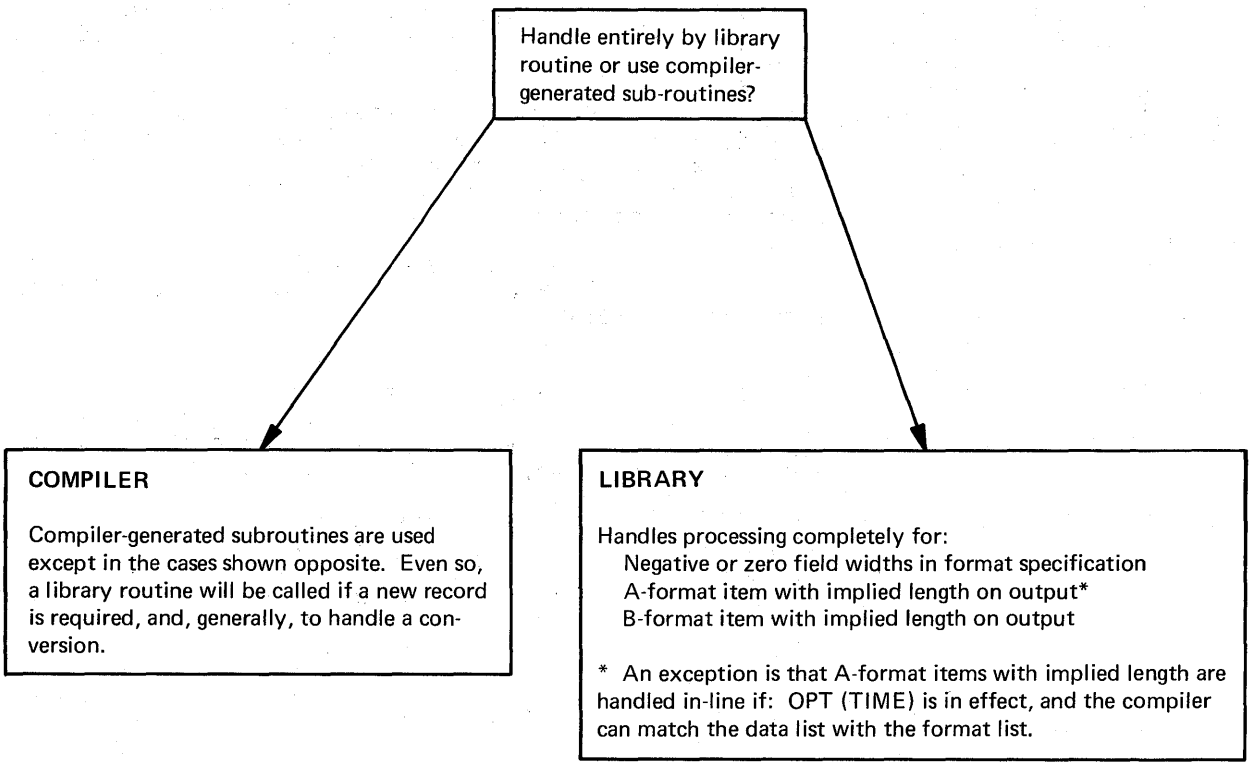


Figure 9.10. Choice of subroutines for edit directed I/O

2. A call to a compiler-generated subroutine to check whether a new record is required, and if so to call a library module to transmit a record by making a call to the transmitter.

   The SIOCB is completed with source or target DEDs and the addresses of the source and the target or their locators.

3. A call to a conversion module or conversion director, or a compiled-code conversion using the information set up in the SIOCB.

4. A further call to a compiler-generated subroutine, to update the buffer control fields, and free any intermediate workspace if spanning was involved.

This sequence is illustrated in the annotated flowchart in figure 9.11. Figure 9.12 shows the code generated for a GET EDIT statement.

Handling Control Format Items

Control format items are implemented by passing the SIOCB, which contains the FED for the control format item, to one of the control format modules. There are four modules:

1. IBMDSPL: library routine for SKIP, PAGE, and LINE formats and options.

2. IBMDSXC: library routine for X and COLUMN formats.

3. IELCGOC: compiler-generated subroutine for X output items that do not span a record boundary.

4. IELCGIA: compiler-generated subroutine for X input items that do not span a record boundary. (This module also has other functions; see the section "Compiler-generated Director Routines" near the end of this chapter.)

Matching and Non-Matching Data and Format Lists

In the majority of edit-directed statements, the data and format lists can be matched during compilation, since the programmer requires conversions for specific variables. However, it is possible to write statements which, because

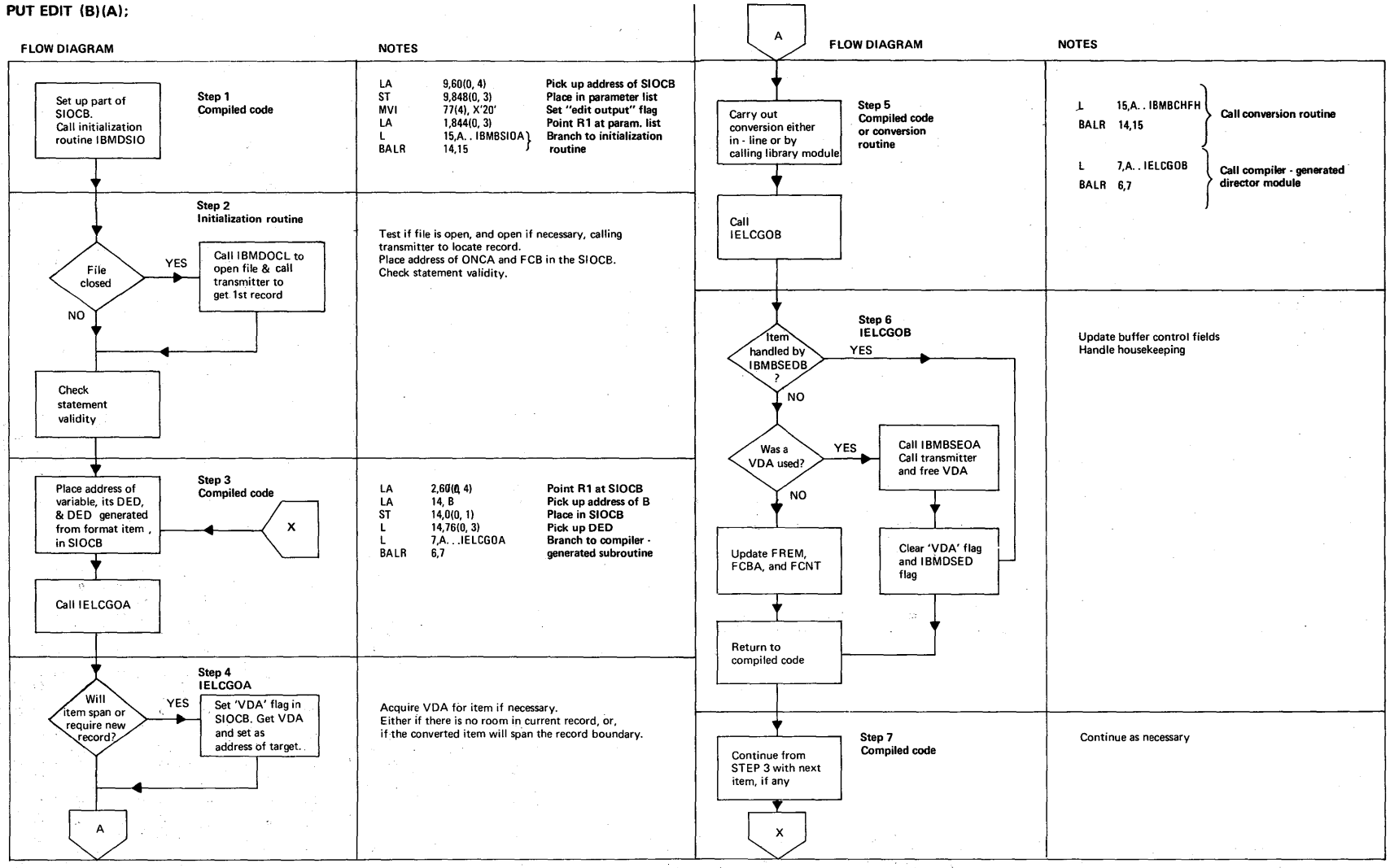


Figure 9.11. Edit directed output statement with matching data and format lists

PL/I statement: GET EDIT (A,B)(F(3),X(8));

```
* STATEMENT NUMBER  3
00007A  41 90 4 010                LA     9,16(0,4)        Pick up address of SIOCB
00007E  50 90 3 060                ST     9,96(0,3)        Store in parameter list
000082  96 80 3 060                OI     96(3),X'80'      Mark end of parameter list
000086  92 24 4 021                MVI    33(4),X'24'      Set 'edit input flag
00008A  41 00 3 050                LA     0,80(0,3)        Set up abnormal locate return address (CL 2)
00008E  50 00 4 028                ST     0,40(0,4)        Store in SIOCB
000092  41 10 3 05C                LA     1,92(0,3)        Point R1 at parameter list
000096  58 F0 3 030                L      15,A..IBMBSIIA   Call stream I/O initialization routine
00009A  05 EF                      BALR   14,15
00009C  41 E0 D 0A8                LA     14,A             Pick up address of source
0000A0  41 F0 3 040                LA     15,DED..A        Pick up address of source DED
0000A4  41 10 4 010                LA     1,16(0,4)        Point R1 at SIOCB
0000A8  50 10 4 008                ST     1,8(0,4)         Save A(SIOCB) in temporary storage
0000AC  90 EF 1 008                STM    14,15,8(1)       Store A(target), A(target DED) in SIOCB
0000B0  41 E0 3 044                LA     14,68(0,3)       Point R14 at FED
0000B4  58 70 3 00C                L      7,A..IELCGIA     Call compiler generated subroutine
0000B8  05 67                      BALR   6,7
0000BA  58 F0 3 02C                L      15,A..IBMBSFIA   Call conversion director
0000BE  05 EF                      BALR   14,15
0000C0  58 70 3 010                L      7,A..IELCGIB     Call compiler generated subroutine
0000C4  05 67                      BALR   6,7
0000C6  41 E0 3 04A                LA     14,74(0,3)       Pick up FED of format item
0000CA  58 10 4 008                L      1,8(0,4)
0000CE  58 70 3 00C                L      7,A..IELCGIA     Call compiler generated subroutine
0000D2  05 67                      BALR   6,7
0000D4  41 E0 D 0AC                LA     14,B             Pick up address of second item
0000D8  50 E0 1 008                ST     14,8(0,1)        Store in SIOCB
0000DC  41 E0 3 044                LA     14,68(0,3)       Point R14 at FED
0000E0  58 70 3 00C                L      7,A..IELCGIA     Call compiler generated subroutine
0000E4  05 67                      BALR   6,7
0000E6  58 F0 3 02C                L      15,A..IBMBSFIA   Call conversion director
0000EA  05 EF                      BALR   14,15
0000EC  58 70 3 010                L      7,A..IELCGIB     Call compiler generated subroutine
0000F0  05 67                      BALR   6,7
0000F2                     CL.2    EQU    *                Abnormal-locate return address
```

Figure 9.12. Typical code generated for a GET EDIT statement

of iteration factors, cannot be matched at compile time. For example, in the statement:

```
PUT EDIT (A,B,C)(N(F(3)),X(10));
```

it is impossible to know at which point the ten-character space indicated by "X(10)" will be required, without knowing the value of N. If the statement had been

```
PUT EDIT (A,B,C)(F(3),X(10));
```

the code would be compiled in the order: handle the conversion of a variable, handle an X item, handle the conversion of a variable, etc., until the data list was exhausted. However, as it is not known at which point the X items will be required in the unmatched statement, it is impossible to compile sequential code to handle the statement. Consequently, the code for each item is compiled separately, and branches are made between the two types of code as the values of the repetition factor indicates. In the example above, the branches would be made when the F item had been executed N times, and when the X item had been executed once.

The code sequences used for matching and non-matching data and format lists are shown in figure 9.13.

Choice of Initialization Routines

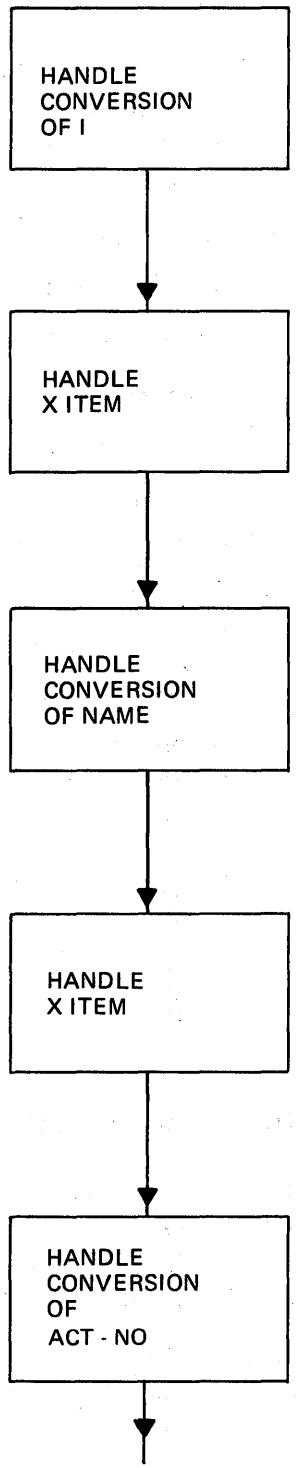
Three initialization routines are available for stream I/O using the FILE option. (The module for use with the STRING option is described later in the chapter.) The initialization routines are:

```
IBMDSII - input only
IBMDSIL - input and output, provided
          the COPY option is not used
IBMDSIO - output only
```

IBMDSIL has space saving advantages when either input or input and output are used in a program. If only output is used, IBMDSIO is the most economical module.

MATCHING LISTS

PUT EDIT (I, NAME, ACT. NO)
(F (3),X (3), A (15), X (3), P'ZZZ9');



UNMATCHING LISTS

PUT EDIT (AB, C, D) ((N) F (3), SKIP, A (4));

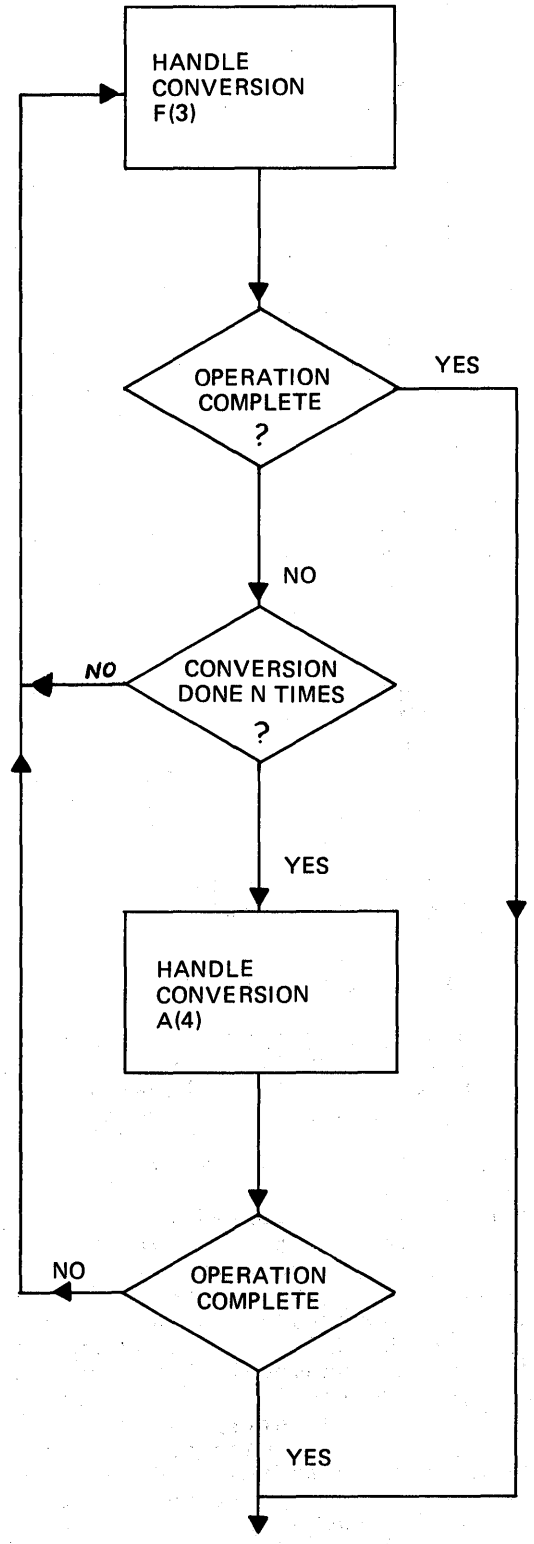


Figure 9.13. Code sequences for matching and non-matching data and format lists

Compiled code calls these modules as follows.

IBMDSIL is used when both stream input and stream output are used in one program and the COPY option is not used for the input.

IBMDSII is used when input with the COPY option is used in the program.

IBMDSIO is used when only output is required, or when both input and output are required and the input uses the COPY option.

The autolink feature of the DOS linkage editor prevents duplicate modules being incorporated into one program. To allow the choice of modules to be made by the linkage editor, IBMDSII contains additional entry points with the same names as those in IBMDSIL. Similary, IBMDSIL contains entry points with the same names as those in IBMDSIO. (The autolink feature resolves external references alphabetically. Thus if there is a reference to IBMDSII, any references to IBMDSIL entry points will be resolved to the entry points in IBMDSII. Similarly, any references to IBMDSIO can be resolved in IBMDSIL.)

Handling Format Options

Format options are handled by a call to the appropriate entry point of the initialization routine.

The initializing module calls the formatting module IBMDSPL to carry out the formatting.

Input and Output of Complete Arrays

When transmitting complete arrays, it is uneconomical for a return to be made to compiled code after each item has been handled. Accordingly, the list- and data-directed director modules have a facility that enables them to handle complete arrays. The modules access the array multipliers, and handle the indexing from information held in the array descriptors. For edit-directed I/O, each element is handled separately.

Effects of the LIMSCONV Option

GET LIST and GET DATA statements do not specify the format of the input in the data stream. Consequently the compiler must allow for any valid input form. Modules to handle the necessary conversions must, therefore, be included in the object module. This frequently results in a number of lengthy modules being link edited but never used.

To overcome this problem, the compiler option LIMSCONV can be used. LIMSCONV specifies that the input stream will not contain items whose format would require the conversions shown below.

Source (input stream)	Target (variable)
arithmetic constant	string
bit string constant	arithmetic
binary constant	arithmetic

The occurrence of input that would cause these conditions results in the raising of the CONVERSION condition.

When the LIMSCONV option is in force, the compiler does not generate external references for the modules that would handle the prohibited conversions. This results in considerable space savings. Compiled code differs when LIMSCONV is in force, in that calls are made to different director modules. IBMDSLJ is called for list-directed statements, and IBMDSDJ for data-directed statements. The list-directed director module, IBMDSLJ, contains code to check for the prohibited conversions and raise the CONVERSION condition if necessary. The data-directed module, IBMDSDJ, is exactly the same as the normal director module, IBMDSDI, except that it calls IBMDSLJ and not IBMDSLI. (Figure 9.8 shows how, in data-directed I/O, the list-directed director module is called to handle the conversions.)

PL/I Conditions in Stream I/O

The following errors and PL/I conditions are particularly relevant to the implementation of stream I/O: TRANSMIT, CONVERSION, NAME (data-directed only), ENDFILE, and unexpected end of file. Unexpected end of file occurs when the end of file is reached in the middle of an input item. Other conditions that occur present no special problems.

TRANSMIT Condition

The rules for raising the TRANSMIT condition in stream I/O are that the

condition shall be raised after the assignment of the potentially incorrect data item. Thus TRANSMIT can be raised on input for a data item even though the transmitter has not been called for the statement involved.

When the TRANSMIT condition is detected by the LIOCS routines, control returns to the error routine in the transmitter, which sets a flag in the FCB indicating a transmission error. The director module inspects this flag, and, if it is set, sets a flag in the SIOCB. For input, TRANSMIT is raised for every item that is taken from a record in the block with which the transmission error was associated. TRANSMIT is raised after the potentially incorrect value has been assigned. For output, TRANSMIT is raised by the transmitter immediately it occurs.

A special entry point, IBMBSEIT, is used by the compiler-generated subroutines to raise the TRANSMIT condition. When called by this entry point, the module calls the error handler with the appropriate error code for the TRANSMIT condition.

CONVERSION Condition

The CONVERSION condition is detected by the conversion modules in the PL/I library. (Conversions that could cause the CONVERSION condition are not handled in-line except where "NOCONVERSION" is specified.) CONVERSION is raised by calling a special library module, IBMDSCV. This module analyzes the type of conversion error, and calls the error handler with an appropriate error code. The module also saves the field that caused the conversion; it is necessary to do so, because the field could be lost if an on-unit was entered and another GET statement executed which resulted in a new record being acquired.

NAME Condition

The NAME condition can occur only in data-directed input. It is raised by the data-directed director module when it cannot find a symbol table to match the name read in. DATAFIELD is set up, and the file positioned for the next read, before calling the error handler, with the appropriate error code

ENDFILE Condition and Unexpected End of File

End of file is detected by the LIOCS routines, which then enter the EOFADDR routine in the transmitter. This routine sets a flag in the FCB. On return to the director modules, the flag is inspected and, depending on the situation in which the transmitter was called, ENDFILE or unexpected end of file is raised by calling the error handler, with the appropriate error code.

For unexpected end of file, the ERROR condition is always raised as soon as the end of file is detected. ENDFILE, in the case of list- and data-directed I/O, is not raised until a further attempt is made to read the input file.

Built-In Functions in Stream I/O

The built-in functions that are relevant to stream I/O are COUNT, DATAFIELD, ONCHAR, and ONSOURCE.

ONCHAR and ONSOURCE are dealt with in chapter 10, under the heading "Raising the CONVERSION Condition."

The COUNT built-in function is handled by the director routines. A count of transmitted items is kept in the FCB; the number is updated after every transmission to or from a PL/I variable.

The DATAFIELD built-in function is handled by the director routine, which places the address of a string locator descriptor for the data in the ONCA. The offending field is first moved to a workspace area, as the buffer may get lost if further stream I/O operations take place in an on-unit.

COPY Option

Implementation of the COPY option involves the use of a pointer, FCPM, in the FCB. FCPM points to the start of the data that is to be copied. Use is also made of the buffer control pointer, FCBA, to point immediately beyond the end of the data that is to be copied. At the end of the GET statement, or when further processing will result in the data to be copied no longer being held between FCPM and FCBA (for example when the transmitter is being called to acquire a new record), the copy module IBMDSCP is called. IBMDSCP moves

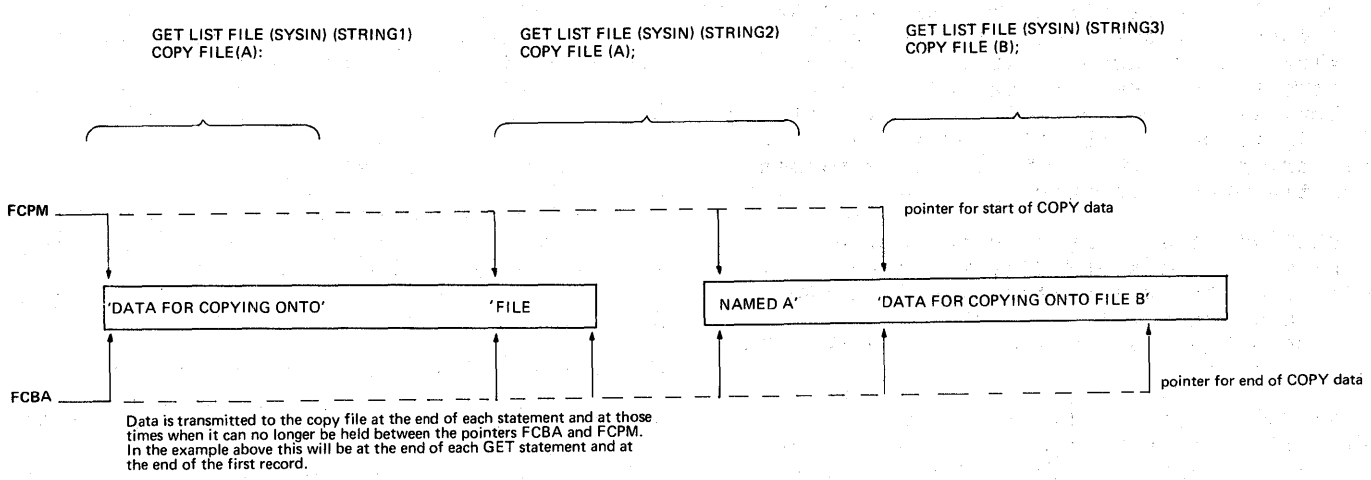


Figure 9.14. Use of FCBA and FCPM in copy option implementation

the data into the buffer being used by the copy file, and then calls the transmitter to transmit the data.

The data to be copied is normally held between FCPM, which is initialized for a statement with the COPY option, and FCBA, which is updated with every input to point to the next position in the buffer from which data will be acquired. This is illustrated in figure 9.14.

When execution of a GET statement with the COPY option is begun, the flag FCOP in the FCB is set, and the address of the FCB for the copy file is placed in the FCB of the input file to which the statement refers.

To ensure that no data is lost, the copy flag, FCOP, is tested in the following situations:

1. By the input transmitter, before a new record is acquired.
2. By the stream I/O initialization/ termination routine, IBMDSII, after the completion of a GET statement with the COPY option.
3. By the close routine, when the input file is being closed.
4. By the initialization routine, IBMDSII every time a GET statement is to be executed.

If the copy flag is on, a call is made to the copy module, which transmits the data directly to the copy file, calling a suitable PL/I output transmitter. The copy flag is turned off after calling IBMDSCP in situations 2, 3, and 4 above.

STRING Option

Since the stream I/O director modules and conversion routines are primarily concerned with moving data in main storage, they are used to implement the STRING option as well as normal stream I/O. However, as the same modules are used, something must be done to prevent calls to a transmitter. It is achieved by having a special STRING module, IBMDSIS, that sets up a dummy FCB containing addresses which result in control being passed to suitable code when an attempt is made to call the transmitter.

Compiled code passes the string initialization module an extended SIOCB in which the dummy FCB is set up. The buffer control fields FCBA and FREM in the dummy FCB are set up as if the string were a

record. The address that would hold the transmitter in the dummy FCB is set up to point to fields that will result in the correct action being taken if an attempt is made to read or write beyond the end of the string, or if a transmitter call is made.

When an attempt is made to call the transmitter for a PUT statement, the address in the transmitter field will have been initialized to point to the error handler. As register 1 will have been pointed to the head of the FCB by the caller, the error code for exceeding string size is placed at the head of the FCB, and the correct error is automatically raised when the branch is made.

When an attempt is made to call a transmitter for a GET STRING statement, the address in the transmitter field is the address of code set up in the dummy FCB that sets the end-of-file flag and returns to the caller.

As far as the caller is concerned, attempting to read beyond the end of the string is equivalent to finding an end-of-file mark in stream I/O statements. Where the ENDFILE condition or unexpected end-of-file would be raised for a stream file, the ERROR condition is raised, and a 'GET STRING SIZE EXCEEDED' message is issued.

Completing String-Handling Operations

In certain circumstances a further call is made to the string routine IBMDSIS, to complete the operation.

For output into a fixed-length string, the routine is called, after the first assignment only, to blank out any remaining bytes in the string. For varying strings, a call is made after every assignment to update the current length of the string.

For input, for varying strings only, the routine is called to update the string information held in the dummy FCB, as this information may have been changed by an assignment to the string.

The need to make a further call to IBMDSIS is flagged in the SIOCB when IBMDSIS is first called in connection with any particular statement. The library director routines and the compiler-generated subroutines test this flag, and call IBMDSIS if necessary.

Summary of Subroutines Used

This section gives a summary of the subroutines used in the implementation of stream-oriented input/output. Detailed descriptions of the library modules are given in the relevant program logic manuals.

Nine different types of subroutine are used in stream I/O. They are:

1. Initializing modules
2. Director modules
3. Transmitter modules
4. Formatting modules
5. Conversion modules
6. External conversion director modules
7. The conversion fix-up module (IBMDSCV)
8. The copy module (IBMDSCP)
9. The string module (IBMDSIS)

Conversion modules are described in chapter 10 of this manual. The other types of module are dealt with below.

INITIALIZING MODULES

Initializing modules initialize the stream I/O statement. There are two of these modules:

IBMDSII - input initializer

IBMDSIO - output initializer

IBMDSII and IBMDSIO are described earlier in this chapter.

DIRECTOR MODULES

Library Director Routines

IBMDSLI - list-directed input

Entry point A: element item
Entry point B: complete array

IBMDSLJ - list-directed input with LIMSCONV

Entry Point A: element item
Entry Point B: complete array

IBMDSLO - list-directed output

Entry point A: element item
Entry point B: complete array

IBMDSDI - data-directed input

Entry point A: with data list
Entry point B: all known variables

IBMDSDJ - data directed input with LIMSCONV

Entry Point A: with data list
Entry Point B: all known variables

IBMDSDO - data-directed output

Entry point A: element variables and whole arrays
Entry point B: single array elements
Entry point C: all known variables and SIGNAL CHECK when CHECK without a check list is enabled.
Entry point D: CHECK output
Entry point T: output a final semicolon

Modules used with Compiler-Generated Subroutines

IBMDSEI - edit-directed input

Entry point A: housekeeping for input item spanning a record boundary.
Entry point T: raise TRANSMIT for input item

IBMDSEO - edit-directed output

Modules for Complete Library Control of Edit-Directed I/O of a Single Item

IBMDSED - Used when complex data or format items appear in the program.

Entry point A: input
Entry point B: output

IBMDSEE - Used when edit-directed input or edit-directed input and output are required in the same program, provided there are no complex items in the program.

Entry points:

IBMBSEEA: Edit-directed input of a data item
IBMBSEHA: Edit-directed output of a data item
IBMBSXCA: X format input
IBMBSXCB: X format output
IBMBSEHB: X format output
IBMBSXCC: COLUMN format input
IBMBSXCD: COLUMN format output
IBMBSEHC: COLUMN format output

IBMDSEH - Used when only output is required in a program, and there are no complex items.

Entry points:

IBMBSEHA: Edit-directed output of a data item
IBMBSEHB: X format output
IBMBSEHC: COLUMN format output

Compiler-Generated Director Routines

For input:

IELCGIA - provides the address of the source of an edit-directed data or X-format item.

IELCGIB - completes the transmission of an edit-directed data item, by freeing a VDA if one was used, updating the COUNT built-in function value, and calling IBMBSEIT if TRANSMIT has been raised.

For output:

IELCGOA - provides the address of the target of an edit-directed data item.

IELCGOB - completes the transmission of an edit-directed data item, updating the buffer items in the FCB, counting the data item, and freeing a VDA if one was used.

TRANSMITTER MODULES

The actual movement of the data between the external medium and the buffer area is carried out by a series of seven transmitter modules, which interface with the LIOCS routines of DOS data management. These modules essentially complete the setting up of the DTF, and issue the data management GET and PUT macro instructions, thus reading or writing one record.

One module is used for input, six for output. The output modules are divided into two groups: one group for PL/I print files, the other for all other output files. Both output module groups contain three modules: one for F-format records,

one for V-format records, and one for U-format records. All modules interface with the queued sequential access method.

The following transmitters are used:

IBMDSTI - input transmitter

IBMDSOF - output transmitter for F-format records

IBMDSOV - output transmitter for V-format records

IBMDSOU - output transmitter for U-format records

IBMDSTF - print transmitter for F-format records

IBMDSTV - print transmitter for V-format records

IBMDSTU - print transmitter for U-format records

The modules IBMDSTI (stream input) and IBMDSTF (stream output for F-format print files) are held in the resident library and are link-edited. All other transmitter modules are held in the transient library and loaded during file opening.

FORMATTING MODULES

Formatting modules control the position of the data on the external medium. There are three formatting modules: two library subroutines, and one compiler-generated subroutine.

Library Subroutines

IBMDSPL - PAGE, LINE, and SKIP format items and options

Entry point A: PAGE option or format item
Entry point B: LINE option or format item
Entry point C: SKIP option or format item

IBMDSXC - X and COLUMN format items

Entry point A: X format input
Entry point B: X format output
Entry point C: COLUMN format input
Entry point D: COLUMN format output

Compiler-Generated Subroutine

IELCGOC - X items, in edit-directed output, that do not span a record boundary.

EXTERNAL CONVERSION DIRECTOR MODULES

The following external conversion director routines are used exclusively in edit-directed I/O:

IBMBSAI - input A, B, and P character formats
IBMBSAO - output A, B, and P character formats
IBMBSCI - input C format
IBMBSCO - output C format
IBMBSFI - input F and E formats
IBMBSFO - output F and E formats
IBMBSPI - input P format arithmetic
IBMBSPO - output P format arithmetic

MISCELLANEOUS MODULES

The other subroutines used in stream I/O are:

IBMDSCP - the copy module

IBMDSIS - the string module

IBMDSCV - the conversion fix-up module

IBMDSMW - module for calculating output format widths not specified in program

Chapter 10: Data Conversion

Note on Terminology

In this chapter, the terms source and target are used when referring to transfer of data. The source is the point from which the data is taken; the target is the point to which it is moved, possibly in a converted format.

The PL/I language specifies situations in which conversion of data types will be carried out. These include the execution of stream I/O and assignment statements, and the evaluation of expressions that include different types of data. The large number of data types allowed in the PL/I language means that some 170 types of conversion are possible. How these conversions are handled by the PL/I Optimizing Compiler depends, to some extent, on the optimization specified for the program.

If no optimization is specified, all except simple conversions are carried out by calls to the library conversion package. If the program is being optimized, all possible conversions are done in-line.

This chapter describes the library conversion package and explains how in-line conversions are handled. It concludes with a description of how the CONVERSION condition is raised.

Before conversions can be understood, knowledge of the way in which data types are held is necessary. This is summarized in figure 10.1.

The Library Conversion Package

The library conversion package consists of 26 modules and is capable of handling all the conversions that are allowed in the DOS PL/I Optimizing Compiler implementation of the PL/I language. All but seven of the modules convert data from one data type to another. As there are approximately 170 possible conversions and only 19 conversion modules, many conversions are done by using a series of modules. For instance, to convert from fixed-decimal to bit-string involves an intermediate conversion to floating-point. The conversion package also contains five control and utility modules, and two modules used for stream I/O.

Data attributes	Stored internally as
BIT(n)	Aligned: one byte for each group of eight bits or part thereof. Unaligned: as many bits as are required, regardless of byte boundaries.
BIT(n) VARYING	As BIT(n), with two-byte prefix containing current length of string.
CHARACTER(n)	One byte per character.
CHARACTER(n) VARYING	As CHARACTER(n), with two-byte prefix containing current length of string.
FIXED DECIMAL(p,q)	Packed decimal: $^1/_2$-byte per digit, plus $^1/_2$-byte for sign.
FIXED BINARY(p,q)	p <= 15: halfword p>15: fullword
FLOAT DECIMAL(p)	p<=6: short floating-point p>6: long floating-point
FLOAT BINARY(p)	p<=21: short floating-point p>21: long floating-point
PICTURE	One byte for each picture character (except K and V)

Figure 10.1. Internal forms of data types

The stream I/O modules move character and bit strings between the data management buffer and the PL/I variable when no conversion is necessary.

A full description of the routines in the library conversion package is given in the publication DOS PL/I Resident Library: Program Logic.

The conversion paths followed for every conversion are known to the compiler, and ESD records are generated for all the modules that will be used. In certain cases, however, the data types involved are not known at compile time. Examples of this are data-directed and list-directed input, and edit-directed input or output when format and data lists cannot be matched. In such cases, the compiler generates ESD records for all conversion modules that could possibly be needed.

SPECIFYING A CONVERSION PATH

When a number of conversion modules need to be used for a certain conversion, it is necessary for there to be some control of the path taken after the first module has been entered. The method used is for each module to have a number of entry points. Each one is entered for a certain type of conversion, and each one implies the subsequent entry points to be invoked for that particular conversion. For instance, the module IBMBCE handles fixed-decimal to fixed-binary conversions. If the module is entered to carry out this conversion, entry point IBMBCEDX is called. However, if it is only an intermediate stage in a conversion from fixed-decimal to bit-string, the entry point IBMBCEDB will be called. When the conversion to floating-point is completed, the conversion to bit will be carried out by the module IBMBCR.

In addition to the use of various entry points to specify the conversion path to be taken, there are two control modules to handle the conversion paths between character-string and arithmetic data.

HOUSEKEEPING WHEN MORE THAN ONE MODULE IS USED

When more than one conversion module is used in a conversion, a method of minimizing the housekeeping has been evolved. This avoids saving registers and acquiring workspace for each module entered. The same library workspace is used for all modules in a single conversion operation. The first module in the chain saves the registers and acquires workspace; the last module frees the workspace and restores the registers.

A simple method is used to allow each module to test whether or not it is the first to be called. A bit at a fixed offset from register 13 is tested. If the module is the first to be called, this bit will be a bit in the calling procedure's DSA, which is always set to zero. If the module is not the first to be called, the bit will be in library workspace and will have been set to one by the previous module. If the module is the first, library workspace will be acquired in the usual manner. If the module is not the first, a branch will be made around this code.

ARGUMENTS PASSED TO THE CONVERSION ROUTINES

Each conversion routine has a standard set of parameters. These consist of the address of the source and target, and the addresses of the DEDs (data element descriptors) for the source and the target. Arguments are passed in a list addressed by register 1. (The source is the variable or constant that requires conversion; the target is the area where the converted result is to be placed.)

The DEDs are used to describe the data type of the element. Those passed to the library conversion package are set up by compiled code in the constants pool. They are described in chapter 4 and fully mapped in appendix B.

COMMUNICATION BETWEEN MODULES

When the conversion path goes through a series of modules, the address of the final target must be retained until the last module is reached.

Temporary targets are created for the intermediate results, and these are passed on as the source for the next module. When information is passed between two conversion modules, registers are normally used rather than a parameter list.

Temporary DEDs are created for intermediate modules. These are set up in library workspace and are based on the original source DEDs.

In some arithmetic conversions to string, precision data is passed through certain modules that do not themselves need such data.

FREE DECIMAL FORMAT

Because all floating-point data is in binary form, there is no direct

Conversion		Comments and conditions	Optimization	
Source	Target		SIZE disabled	SIZE enabled
Fixed binary	Fixed binary	-	-	-
	Fixed decimal	If either scale factor = 0 and the other factor ≤ 0, the optimization can be 'none'.	time	time
	Floating-point	If source scale factor = 0, the optimization can be 'none' (whether SIZE is enabled or not).	time	time
	Bit string	String must be fixed-length, aligned, and with length ≤2048.	-	not done in-line
	Character string or picture	Source scale factor must be ≤ 0. String must be fixed-length with length ≤256. Picture type 1, 2, or 3.	time	not done in-line
Fixed decimal	Fixed binary	If source and target scales have the same sign and are non-zero, the optimization (SIZE disabled) must be 'time'.	-	time
	Fixed decimal	-	-	-
	Floating-point	Source precision must be <10.	time	time
	Bit string	Source scale factor must be zero. String must be fixed-length, aligned, and with length ≤2048.	-	not done in-line
	Character string	Source scale factor must be ≥ 0. String must be fixed-length and length ≤256.	time	time
	Picture	Picture type 1, 2, or 3. For picture types 1 and 2 with no sign, optimization can be 'none'.	time	not done in-line
Floating-point	Fixed binary	-	time	not done in-line
	Fixed decimal	Target precision must be ≤9.	time	not done in-line
	Floating-point	Source and target may be single or double length.	-	-
	Bit string	String must be fixed-length, aligned, and with length ≤2048.	time	not done in-line

Figure 10.2. (Part 1 of 2). Data conversions performed in-line

representation of the PL/I floating-point decimal format. In order to simplify certain conversions, a simulated floating-point decimal format is employed by the optimizing compiler. This format is termed free decimal (sometimes known as packed intermediate decimal). The format of free decimal is a 17-digit packed decimal mantissa and a fullword binary exponent. Conversions to and from free decimal form an integral part of the arithmetic conversion package.

Conversion		Comments and conditions	Optimization	
Source	Target		SIZE disabled	SIZE enabled
Bit string	Fixed binary	Source string must be fixed-length, aligned, and with length ≤2048.	-	not done in-line
	Fixed decimal and floating-point	Source must be fixed-length, aligned, and with length <32.	time	not done in-line
Picture	Character string	String must be fixed-length with length ≤256.	-	-
	Picture	Pictures must be identical.	-	-
Picture type 1	Fixed binary	Source precision must be <10.	time	not done in-line
	Fixed decimal	If picture has a sign, the optimization must be 'space'.	-	not done
	Floating-point	Source precision must be <10.	time	not done
	Picture	Picture type 1, 2 or 3.	time	not done in-line
Locator	Locator	-	-	-
Label	Label	-	-	-

The word "time" in the columns headed "Optimization" indicates that the conversion is done in-line only if optimization has been specified; "not done in-line" indicates that the conversion is done by library call.

Figure 10.2. (Part 2 of 2). Data conversions performed in-line

In-Line Conversions

The optimizing compiler generates in-line code for the more commonly used conversions. Eighteen basic types of conversion are handled in-line. Several of these basic types are used in conjunction, to enable a total of 28 conversions to be handled in-line. The circumstances in which in-line conversions are used are shown in figure 10.2.

An example of the way in which a compiler conversion is used to convert from fixed-binary to fixed-decimal is given below. A list of the eighteen fundamental compiler conversions is given in figure 10.3.

Note on Picture Types

Figures 10.2 and 10.3 use the terms "Picture type 1, 2, and 3". These picture types must contain only the following characters:

V and 9

Drifting or non-drifting characters $, +

Zero suppression characters Z *

Punctuation characters, . / B

The types are defined as follows.

Picture type 1: Pictures of all 9s with (optionally) a V and a leading or trailing sign. For example:

'99V999', '99', 'S99V9', '99V+', '$999'

Conversion number	Conversion
2	Fixed-binary to floating-point
3	Floating-point to fixed-binary
4	Fixed-decimal to floating-point
5	Floating-point to fixed-decimal
6	Fixed-binary to fixed-decimal
7	Fixed-decimal to fixed-binary
8	Character-string to fixed-decimal
9	Character-string to floating-point
10	Character-string to fixed-binary
12	Fixed-decimal to character-string
14	Bit-string to character-string
15	Fixed-binary to bit-string
16	Floating-point to bit-string
17	Bit-string to fixed-binary
18	Fixed-decimal to picture type 1
19	Fixed-decimal to picture type 2
20	Fixed-decimal to picture type 3
21	Picture type 1 to fixed-decimal

Note: Conversions numbers 1, 11, and 13 not used.

Figure 10.3. Fundamental in-line conversions

Picture type 2: Pictures with zero suppression characters and (optionally) punctuation characters and a sign character. Also, type 1 pictures with punctuation characters. For example:

```
'ZZZ', '**/**9', 'ZZ9V.99', '+ZZ.ZZZ',
  '$///99', '9.9'
```

Picture type 3: Pictures with drifting strings and (optionally) insertion characters and a sign character. For example:

```
'$$$$', '-,--9', 'S/SS/S9',
  '+++9V.9','$$$9-'
```

Sometimes a picture conversion is not performed in-line even though the picture is one of the above types. This may be because:

1. There is no overlap between the digit positions in the source and target. For example:

   DECIMAL (6,8) or DECIMAL (5, -3) to PIC '999V99' will not be performed

2. The picture may have certain characteristics that make it difficult to handle in-line. For example:

   a. Punctuation between a drifting Z or a drifting * and the first 9 is not preceded by a V. For example:

      'ZZ.99'

   b. Drifting or zero expression characters to the right of the decimal point. For example:

      'ZZV.ZZ', '++V++'

BASIC CONVERSIONS

Example: Fixed-Binary to Fixed-Decimal (Compiler Conversion No. 6)

The conversion is performed by converting from binary to decimal via a CVD instruction, with a scale-matching operation (to line up the decimal and binary points) either before or after the CVD (or occasionally both). This scale-matching operation is done by shifts where possible but, depending on scales and precision, a decimal multiplier is sometimes used.

```
DCL SOURCE FIXED BINARY (31,9),
    TARGET FIXED DECIMAL (15,-6);
TARGET=SOURCE;
```

```
L      R1,SOURCE                    Load source in general register.
LTR    R1,R1                        Determine sign of source.
BNM    Compiler-label               Branch if >=0.
A      R1,CONST                     Add a constant to negative source, rounding toward
                                    zero before subsequent divide (right shift).
Compiler-label EQU *
SRA    R1,9                         Divide by source scale (2**9).
CVD    R1,WSP                       Convert to decimal in workspace.
ZAP    TARGET(8),WSP(5)             Transfer to target, at the same time dividing by
                                    10**6.
MVN    TARGET+7(1),WSP+7            Transfer the sign.
```

MULTIPLE CONVERSIONS

The conversions listed in figure 10.3 can be regarded as fundamental types. A number of other conversions can be performed by using two fundamental conversions in series. These are shown in figure 10.4.

HYBRID CONVERSION

Finally, there is one hybrid conversion that is carried out partially in-line. This is floating-point to character-string, which requires an interpretive routine to analyze the floating-point data (as distinct from the attributes, which all the others use), in order to generate the correct scale factor. This is done by the library, because in-line code would use the same algorithm. However, partial optimization is carried out by setting up a character string of the correct length before calling the library, and then handling the subsequent string assignment in-line.

Raising the Conversion Condition

The PL/I language specifies that when an invalid conversion is attempted on character-string data, the CONVERSION condition will be raised unless it has been disabled.

When the CONVERSION condition has been raised, the language allows the program to access the invalid field or character by use of the ONSOURCE or ONCHAR built-in function. The language also stipulates that conversion should be attempted again if an on-unit is entered in which the ONSOURCE or ONCHAR pseudovariable is used to change the invalid field or character.

Raising the CONVERSION condition

Conversion required	Compiler conversions used
Fixed-decimal to bit-string	No. 7 Fixed-decimal to fixed-binary No. 15 Fixed-binary to bit-string
Floating-point to bit-string	No. 3 Floating-point to fixed-binary No. 15 Fixed-binary to bit-string
Bit-string to fixed-decimal	No. 17 Bit-string to fixed-binary No. 6 Fixed-binary to fixed-decimal
Bit-string to floating-point	No. 17 Bit-string to fixed-binary No. 2 Fixed-binary to floating-point
Character-string to bit-string	No. 10 Character-string to fixed-binary No. 15 Fixed-binary to bit-string
Fixed-binary to character-string	No. 6 Fixed-binary to fixed-decimal No. 12 Fixed-decimal to character-string
Fixed-binary to decimal picture	No. 6 Fixed-binary to fixed-decimal No. 18, 19, or 20 Fixed-decimal to picture
Floating-point to decimal picture	No. 5 Floating-point to fixed-decimal No. 18, 19, or 20 Fixed-decimal to picture
Decimal picture to fixed-binary	No. 21 Picture to fixed-decimal No. 7 Fixed-decimal to fixed-binary
Decimal picture to floating-point	No. 21 Picture to fixed-decimal No. 4 Fixed-decimal to floating-point
Decimal picture to decimal picture	No. 21 Picture to fixed-decimal No. 18, 19, or 20 Fixed-decimal to picture

Figure 10.4. Multiple conversions

involves a number of housekeeping problems, which are handled by a special conversion module, IBMBSCV. IBMBSCV is never called by compiled code, since conversions that could raise the CONVERSION condition are not attempted in-line unless the CONVERSION condition is disabled. IBMBSCV produces the correct error code for the error handler, IBMDERR, and looks after the housekeeping problems.

The alternative to using a separate housekeeping module would be to place the code either in the error handler or in the various conversion modules. These solutions would result in a considerable overhead being carried either by all types of errors or by all correct conversions. The reason for the overhead lies principally in the facility offered by the language of using the ONSOURCE and ONCHAR built-in functions to access and optionally change the field causing the error, and subsequently reattempting the conversion on the changed field.

Before any conversion in which the CONVERSION condition could be raised is attempted, the ONSOURCE field in the ONCA must be set up, and the address at which a reattempted conversion should begin must also be placed in the ONCA.

The code carrying out the conversion must then test the validity of the field to be converted and, if it is invalid, set the ONCHAR field in the ONCA to the first invalid character. The module IBMBSCV is then called to diagnose the conversion and produce the correct error code for the error handler. There are some twenty possible error codes associated with the CONVERSION condition.

If the condition was raised during the execution of stream input, further action is necessary. This is because an on-unit may specify further input, and the buffer which contains the ONSOURCE field may be lost. For example the on-unit might be:

```
ON CONVERSION BEGIN;
ON CONVERSION SYSTEM;  /* PREVENTS
                          RECURSIVE ENTRY*/
   GET LIST (KEYB);
   IF KEYB< 200 THEN ONCHAR ='1';
                 ELSE ONCHAR ='9';
   END;
```

If KEYB was in the next record, the source field that caused the conversion would be lost. To prevent this, a VDA is acquired in the LIFO stack, and the source field is stored in this VDA. The ONSOURCE and ONCHAR pointers are altered to point to the field in the VDA, and all further operations are carried out on this field.

The NAB pointer associated with the block in which the interrupt occurred must then be altered so that it encompasses the VDA. The fact that the NAB pointer has been altered must be known in the block for a GOTO out of block to be handled. The reset-NAB bit is accordingly set to one in the relevant DSA. When these operations are complete, IBMBSCV calls the error-handling module IBMDERR.

Chapter 11: Miscellaneous Library Subroutines and System Interfaces

In addition to employing the PL/I libraries for the functions described in previous chapters, the DOS PL/I Optimizing Compiler calls on a large number of computational and data-handling subroutines and on subroutines that provide interfaces with the operating system for such functions as TIME and DATE. These miscellaneous library calls are discussed in this chapter. The library subroutines themselves are fully described in the publications IBM Disk Operating System: PL/I Resident Library Program Logic and IBM Disk Operating System: PL/I Transient Library Program Logic.

This chapter is divided into two main sections: the first deals with the computational and data-handling subroutines, and the second with miscellaneous system interfaces.

Computational and Data-Handling Subroutines

The computational and data-handling subroutines are used to handle all the mathematical built-in functions, the majority of arithmetic built-in functions, and a number of array-handling, structure-handling, and string-handling functions. The extent to which library calls are used depends on the level of optimization specified by the programmer, the type of data involved, and, for string functions, on whether STRINGRANGE and STRINGSIZE are enabled.

ARITHMETIC AND MATHEMATICAL SUBROUTINES

The compiler always uses library subroutines for mathematical functions. The use of compiled code in these circumstances is impracticable. Where possible, arithmetic functions are handled by in-line code. The circumstances in which library subroutines are used are listed in figure 11.1.

Considerable use is made of chains of library modules to carry out the various functions. For example, the subroutines that handle complex arithmetic normally call on those that handle real values to process each part of a complex number; similarly, the square-root subroutine is used in the computation of several of the trigonometrical functions.

Arguments are passed to the arithmetic and mathematical subroutines either in registers or in a parameter list addressed from register 1. The use of registers results in faster execution, but allows less flexibility in use of the routines. All built-in functions, except the STRING built-in function, have their arguments passed in a list comprising the addresses of the source and target (and sometimes also the addresses of DEDs). Where possible, other routines use registers. Computational routines are always carried out in floating-point unless otherwise indicated. This may involve conversion before calling the routine.

ARRAY, STRING, AND STRUCTURE SUBROUTINES

A number of array, string, and structure subroutines are included in the DOS PL/I Resident Library. These are used to carry out certain of the array and string built-in functions and a number of other operations. Where possible, in-line code is generated to carry out these functions. However, the enablement of STRINGSIZE, the use of unaligned bit strings, and the use of adjustable and certain varying-length strings will result in calls being made to the library sub-routines.

The subroutines involved in these functions are shown in figure 11.2. Two of them, IBMBAIH and IBMBAMM, are concerned with the handling of data aggregates rather than with the execution of specific operations. They are discussed below.

Indexing Interleaved Arrays (IBMBAIH)

IBMBAIH is used to assist the other library array-handling subroutines to process multidimensional interleaved arrays. It is not called by compiled code.

Interleaved arrays are arrays whose elements are not held contiguously in storage. They occur in arrays of structures. For example, the declaration:

Function	Data type	Module name	When used
REAL ARGUMENTS			
Integer exponentiation	Short floating-point	IBMBMXS	When exponent is a variable
	Long floating-point	IBMBMXL	When exponent is a variable
General exponentiation	Short floating-point	IBMBMYS	Always
	Long floating-point	IBMBMYK	Always
COMPLEX ARGUMENTS			
Integer exponentiation	Short floating-point	IBMBMXW	When exponent is a variable
	Long floating-point	IBMBMXY	When exponent is a variable
General exponentiation	Short floating-point	IBMBMYX	Always
	Long floating-point	IBMBMYY	Always

Figure 11.1. Arithmetic operations performed by library subroutines

IBMBAAH	ALL and ANY built-in functions
IBMBAIH	Indexer for interleaved arrays
IBMBAMM	Structure mapping
IBMBANM	STRING built-in function
IBMBAPC	PROD built-in function (fixed-point integer)
IBMBAPF	PROD built-in function (floating-point)
IBMBAPM	STRING pseudo-variable
IBMBASC	SUM built-in function (fixed-point)
IBMBASF	SUM built-in function (floating-point)
IBMBAYF	POLY built-in function (floating-point)
IBMBBBA	AND and OR logical operations (aligned bit strings)
IBMBBBC	Compare aligned bit strings
IBMBBBN	Invert aligned bit string (NOT)

Figure 11.2. (Part 1 of 2). Array, structure, and string subroutines

```
DCL 1 STRUCTURE (2),
        2 A(2),
        2 B ;
```

would result in successive storage locations being allocated to elements of A and B as follows:

```
A(1,1),A(1,2),B(1),A(2,1),A(2,2),B(2)
```

Both A and B are interleaved arrays. A is a two-dimensional array, the first row of which is separated from the second by an element of B. As can be seen, the elements of A are not contiguous, nor is there a fixed interval between their addresses.

IBMBBCI	INDEX built-in function (character string)
IBMBBCK	Concatenate character strings and REPEAT built-in function
IBMBBCT	TRANSLATE built-in function (character string)
IBMBBCV	VERIFY built-in function (character string)
IBMBBGB	BOOL built-in function
IBMBBGC	Compare unaligned bit strings
IBMBBGF	Bit-string assignment (aligned, source and target)
IBMBBGI	INDEX built-in function (bit string)
IBMBBGK	Concatenate bit strings, REPEAT built-in function, and assign
IBMBBGS	Produces SLD (SUBSTR built-in function)
IBMBBGT	TRANSLATE built-in function (bit string)
IBMBBGV	VERIFY built-in function (bit string)

Figure 11.2. (Part 2 of 2). Array, structure, and string subroutines

The interval between the addresses of elements of an interleaved array referred to by varying only the final subscript is always fixed, and these elements can be stepped through by using the last multiplier from the array descriptor. However, such groups of contiguous elements are not themselves necessarily contiguous.

When IBMBAIH is called, it is passed the address of a work area in which to construct a table, the address of the array descriptor, and the number of dimensions in the array. Basically, IBMBAIH calculates the extent of each dimension and enters this information in the table; it then calculates the increments that must be added in order to step between elements that may be non-contiguous (see figure 11.3). The information in the completed table is used by the calling module to address successive elements of the array using simple code.

Structure Mapping (IBMBAMM)

Structures are normally mapped during compilation. However, certain structures that contain adjustable strings or arrays cannot be mapped until the actual lengths or bounds are known. Compiled code calls on the module IBMBAMM to carry out this mapping. There are four entry points:

IBMBAMMA Compute length of structure.

IBMBAMMB Map structure in PL/I manner.

IBMBAMMC Map structure in COBOL manner (for interlanguage comunication or for files declared with the COBOL option).

IBMBAMMD Map structure declared with REFER option.

Miscellaneous System Interfaces

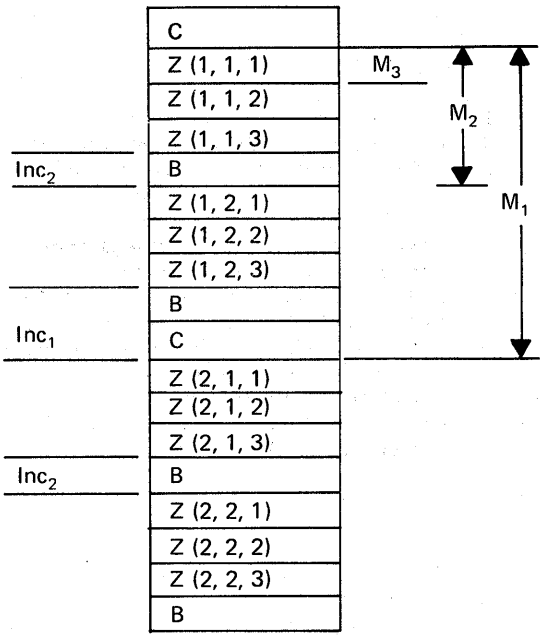
In addition to the system interface used for input and output, the PL/I Optimizing Compiler makes use of a number of other system facilities. These are for the DELAY, DISPLAY, and WAIT statements, the TIME and DATE built-in functions, and the sort/merge and checkpoint/restart built-in subroutines.

Calls to these facilities are made through library subroutines held in the DOS PL/I Resident Library. These subroutines act as an interface, issuing any SVC calls that may be necessary, and handling housekeeping problems. The descriptions of the subroutines in this chapter are kept to a minimum except where the housekeeping problems are large and have a major effect on the contents of main storage. In these cases, background information is given and the various control blocks are explained, thus enabling the situation during execution to be understood.

Declaration

```
DECLARE   1 X(2),
            2 C,
            2 Y (2),
              3 Z (3),
              3 B;
```

Storage



Z is a three-dimensional interleaved array, for which

M_1, M_2, and M_3 = multipliers held in array descriptor (see chapter 4)

Inc_1 and Inc_2 = intervals between addresses of successive elements of Z when subscripts for first and second dimensions, respectively, change

The increment when the subscript for the ith dimension changes is computed as follows:

$Inc_i = M_i - E_{i+1} * M_{i+1} + Inc_{i+1}$

Where E_{i+1} is the extent of the (i+1)th dimension.

Increment table for array Z (as initialized by IBMBAIH)

2nd dimension	2	subscript count
	2	extent of dimension
	Inc_2	increment
1st dimension	2	subscript count
	2	extent of dimension
	Inc_1	increment

Note: IBMBAIH returns the extent of the nth dimension in register 1. (In this example, the extent of the 3rd dimension = 3.)

Figure 11.3. Indexing interleaved arrays

The DOS macro instructions referred to below are described in IBM System/360 Disk Operating System: Supervisor and Input/Output Macros.

TIME

The PL/I TIME built-in function is implemented by issuing a GETIME macro instruction. This is done by the module IBMDJTT.

On entry from compiled code, register 1 points to the address of the character-string target. The module issues a GETIME macro instruction with the TU option, and the current time is returned as a character string of length nine in the form hhmmssttt. The GETIME macro, and consequently this module, returns the time of day to the nearest 1/300 second.

DATE

The PL/I DATE built-in function is implemented by module IBMDJDT, which accesses the job's communications region to obtain the necessary data.

On entry from compiled code, register 1 points to the address of the date character string. The module accesses bytes 0 - 7 of the communications region, which contain the data in the form ddmmyy if bit 0 of the date-convention byte is 1, or in the form mmddyy if the bit is 0. The date is then translated using the appropriate translate table. The date is returned as a character string of length six in the form yymmdd.

DELAY

The PL/I DELAY statement is implemented using the SETIME and WAIT macro instructions, which are issued by module IBMDJDY. The SETIME macro instruction allows the interval time to be set only to an integral number of seconds; hence the delay is restricted to an integral number of seconds. On entry from compiled code, register 1 points to the number of milliseconds delay. The delay interval is rounded to the nearest second and a maximum interval of less than 55919 seconds set up. A SETIME macro is issued, specifying the interval and a timer event control block (TECB) name. A WAIT macro instruction is issued to delay execution for the required interval; register 1, unchanged by the SETIME macro, points to the TECB. On completion of the wait, the time will have elapsed and control is returned to compiled code.

DISPLAY

The PL/I DISPLAY statement is implemented by two library modules, IBMDJDS and IBMDJDZ. IBMDJDZ handles display without the EVENT option; IBMDJDS handles display with the EVENT option.

IBMDJDS DISPLAY with EVENT Option.

DISPLAY without REPLY Option

If no reply is requested, the message is scanned and trailing blanks are removed. An EXCP macro instruction is issued, specifying a channel control block (CCB) that contains the address and length of the message.

If a reply is requested, the message is scanned and trailing blanks are removed. Three channel command words (CCWs) are used for the message and reply: the first to put out the message, the second to put out a standard message saying "awaiting reply," and the third to accept the reply. An EXCP macro instruction is issued; if the EVENT option is not specified, a WAIT macro instruction is also issued. Return is made to compiled code.

If the EVENT option is specified, the event variable is checked before the EXCP macro instruction is issued, to see if it is active. When the corresponding WAIT statement in compiled code is executed, IBMDJWT returns control to IBMBJDSB on completion of the event.

IBMDJDZ - DISPLAY without the EVENT Option

When IBMDJDZ is entered, the display string is scanned and trailing blanks are removed. If there is no REPLY option, an EXCP macro and a WAIT macro are issued to transmit the message to the console. The channel control block (CCB) used contains the address and length of the message. Return is then made to the caller.

If there is a REPLY option, the EXCP and WAIT macros are issued as above. However,

the EXCP specifies a chain of three channel control blocks. The first channel control block is to transmit the display string, the second is to transmit a standard message to the operator stating that a reply is required, and the third is for the reply to the message. When the reply is recieved, return is made to the caller.

If a unit check or exception occurs on the console, any reply string is blanked out and the EXCP and WAIT macros for the display or reply are reissued. The ERROR condition is raised if there is a zero length display with the REPLY option, or if the length of the string to accept the reply is zero.

SORT/MERGE

The PL/I programmer can make use of the DOS sort/merge facilities through a call to the built-in subroutine PLISRT. The method of using the facility is fully described in the publication IBM Operating System: PL/I Optimizing Compiler Programmers' Guide.

The DOS sort/merge program includes a number of user exits that can be conveniently thought of as allowing the programmer to write sections of code that become included in the sort/merge routines. Two of these user exits can be used by the PL/I programmer: user exit 15 allows records to be set up by PL/I and passed to the SORT routines; user exit 35 allows records that have been sorted to be passed to and processed by the PL/I program.

Exits are not allowed in the PL/I language. To overcome this problem, code is inserted between the sort/merge modules and the PL/I routines. A bootstrap module, IBMDKST, is used, and this module acts as an interface between SORT and PL/I. The module retains the PL/I environment and restores it on return from sort/merge so that the PL/I exit-15 or exit-35 code can operate in a PL/I environment. Similarly, it restores the environment for SORT on return from the exit.

Housekeeping Problems

Various housekeeping problems occur in the user exit procedures, since there is no DSA chain through the SORT modules. Particularly difficult is the handling of a GOTO out of the exit procedure that passes control to a procedure on the same or higher level as the procedure that originally called the sort program. This action implicitly terminates SORT. However, SORT will not be terminated by standard PL/I action, since it does not function in the PL/I environment.

The problems are overcome by setting up a chainback that includes a simulated DSA for the SORT routines. This DSA is specially flagged so that it can be recognised by the GOTO code. The chaining of save areas in shown in figure 11.4.

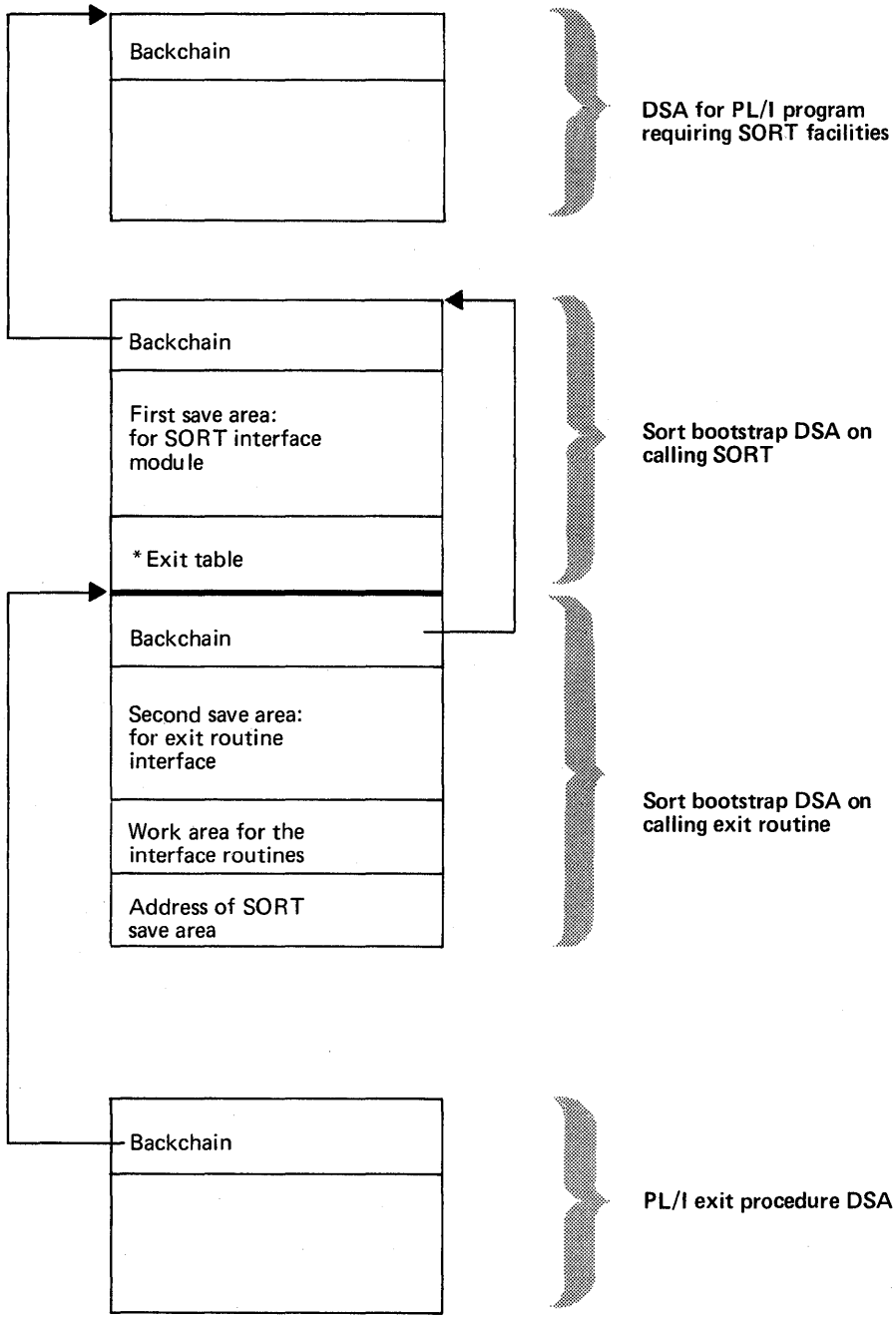
An area of workspace is acquired by the bootstrap routine IBMDKST. This consists of one level of library workspace, a VDA of the correct size to hold two save areas, and a nine-word area of workspace. The second DSA is chained back to the first. (See figure 11.4.)

If the SORT program is terminated by a GOTO out of the block that contains the PL/I exit program, the SORT routine has to be terminated before the GOTO can be completed. This is done by the GOTO routine looking for the SORT exit DSA (which is specially flagged) in the DSA chain. If one is found, a return code of 8 is set up and return made to the SORT routine. This results in the termination of the SORT routine, and the GOTO can then be continued in the usual manner by following the DSA backchain through the bootstrap routine until the target DSA is reached.

For handling on-units in the exit procedure, the DSA chain can be followed without reference to SORT.

Restoration of the PL/I Environment on Exit from SORT

When an exit is made from SORT, it is necessary to restore the PL/I environment. The method used is to have a section of code that restores the registers at the point to which SORT makes its exit. Use is made of the SORT exit table shown in figure 11.4. As can be seen, which ever exit is taken, control passes to this code, which saves the registers passed by SORT and restores the registers of the bootstrap module IBMDKST, thus restoring the PL/I environment. The save area of the SORT bootstrap routine is addressed by means of an offset from the code that is being executed. This is possible because the SORT exit table and the register save area are both held in the same workspace at a fixed offset from each other. The code is not included in the bootstrap module, in order to preserve reentrancy.



*Exit table

	NOP	0	not used
Entry point for E15	BC	15,12(15)	branch to exit code for E15 exit
Entry point for E35	BC	15,12(15)	branch to exit code for E35 exit
	STM	14,12,12(13)	save sort registers
	L	2,28(15)	locate bootstrap save area
	LM	2,12,28(2)	restore bootstrap registers
	B	exit bootstrap	initialized address of routine
	DC	A (save area 1)	address of first save area

Figure 11.4. DSA chaining during execution of SORT

Summary of Work Done by the SORT Module

Before calling the SORT program, IBMDKST:

1. Obtains a VDA for two DSAs.
2. Creates a parameter list suitable for SORT.
3. Sets up addressability code for use after return from SORT.
4. Sets the program check exit so that a program check results in entry being made to a section of the sort bootstrap. The sort bootstrap then determines the error, puts out a message to SYSPRINT indicating that a program check has occurred during the execution of SORT, and then terminates the program.

On exit from the SORT program, the addressability code saves the registers of SORT and reestablishes the PL/I environment, and then branches to an entry point of IBMDKST, which:

1. Resets the program-check exit so that control will pass to the PL/I error-handling routines.
2. Sets up parameters for the PL/I exit routine from information passed by SORT.
3. Calls the PL/I exit routine.

Setting the return code in the PL/I exit program resets the parameters that IBMDKST passes to the SORT routines. (See figure 11.5.)

Storage for SORT

Storage for sort/merge workspace and the modules used is obtained in the LIFO stack. A VDA of the correct length is obtained by the bootstrap module. The length required must be specified in the arguments that are given in the call to PLISORT.

CHECKPOINT/RESTART

The PL/I Optimizing Compiler allows the programmer to make use of the system checkpoint/restart facilities by calling the built-in subroutine PLICKPT. This is implemented by a call to the resident-library subroutine IBMDKCP, which issues the CKPT macro instruction.

Before the CKPT macro instruction is issued, two control blocks must be set up. One of these control blocks contains the names of all tape files that are open; it is used to reposition the tapes on restart. The other control block contains verification information for all disk files that are open; it is used to verify that the disk packs are on the same devices on restart as they were when the check-point was taken. The two control blocks are held in the workspace acquired for the module IBMDKCP.

When a restart is made, control is passed to the module IBMDKCP at a fixed entry point. After carrying out necessary checks, control is then returned to the calling routine in the normal manner. Control is thus returned to the statement after the call to PLICKPT, and processing continues.

WAIT

The PL/I WAIT statement allows the programmer to specify that processing shall halt until a specified number of events are complete. In this implementation, an event can be associated with either a record I/O operation or a DISPLAY statement, or it can be an inactive event that is not associated with any operation.

All information relating to an event is kept in an event variable. This is a control block of five words in length; it is treated for storage allocation like any other PL/I variable. The event variable holds information on whether the event is associated with an operation and whether it is complete; it also records the status of the event (i.e., whether the associated operation was completed successfully or otherwise). When an event is associated with an operation, it is said to be active; otherwise, it is said to be inactive.

When the wait statement is used, the keyword WAIT is followed by a list of events that are to be waited on. A number can follow this list, indicating that only that number of events need be completed before processing can continue. Typical WAIT statements are:

```
WAIT (EVENT1,EVENT2);

WAIT (EVENT1,EVENT2) (1);
```

For the first statement, both the events would have to be completed before processing could continue. For the second statement, processing would continue as soon as either of the events was complete.

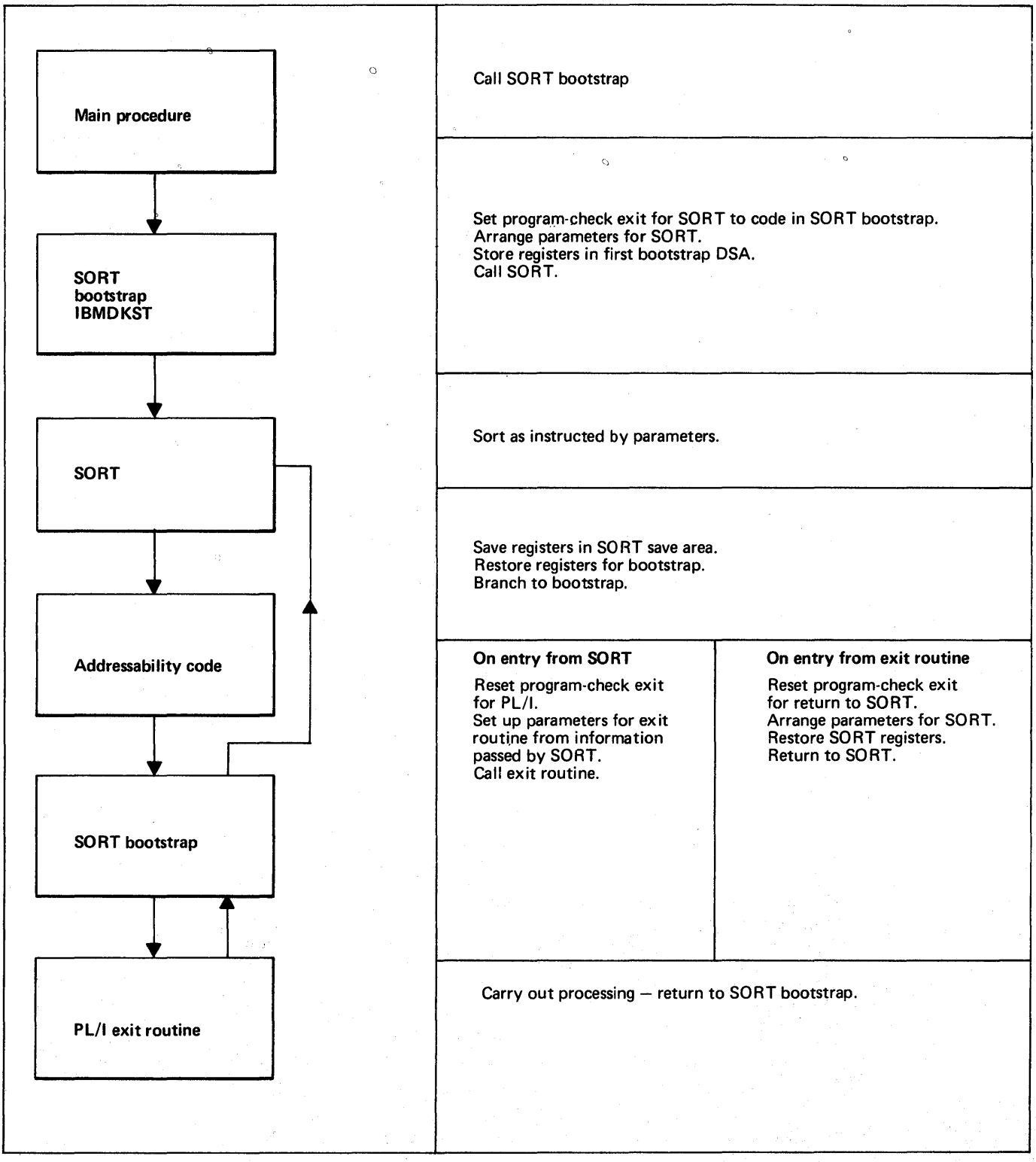


Figure 11.5. Summary of action during use of SORT exit

```
WAITER: PROC OPTIONS (MAIN);

          ON TRANSMIT (A) CALL L;
          ON TRANSMIT (C) CALL L;
          ON TRANSMIT (X) CALL L;

          ON RECORD (A) CALL M;
          ON RECORD (C) CALL M;
          ON RECORD (X) CALL M;
          K=0;
1         READ FILE (A) INTO (B) EVENT
         (E1);
2          READ FILE (C) INTO (D) EVENT
         (E2);
            •
            •
            •
3         WAIT (E1,E2);
            •
            •
            •
4         IF K=1 THEN WAIT (E2);
            •
            •
            •
5 BOOTLE: WAIT (E3);
            •
            •
            •
       L: PROC;
6          COMPLETION (E3)='1'B;
7          GO TO BOOTLE;
           END L;
       M: PROC;
8          COMPLETION (E3)='1'B;
9          WAIT (E2);
10         K=1;
11          READ FILE(X) INTO(Y) EVENT
             (E2);
           END M;

           END WAITER;
```

Figure 11.6. Example of WAIT implementation problems

The WAIT statement implemented in any particular installation depends on whether or not that particular system supports the DOS data-management WAITM macro instruction. If it does not support this macro instruction, the full PL/I WAIT statement cannot be supported and the routine IBMGJWT will be included in the DOS PL/I Resident Library. If the WAITM macro instruction is supported, the full WAIT statement can be supported, and the module IBMDJWT will normally be included in the resident library, although it will be possible to specify the other module if the full WAIT facilities are not required.

The difference between the two modules is:

IBMGJWT Supports only waits on single events.

IBMDJWT Supports waits on multiple events.

Event Variables

When storage is allocated for an event variable, the event variable is set inactive and incomplete. When the EVENT option is used to associate the event with an operation, the event variable is set active and incomplete. When a WAIT statement is executed and the operation associated with the event has been completed, the event variable is set inactive and complete. The status of the event is also set at this time, indicating whether or not the operation was successfully completed.

The PL/I language allows the programmer to set complete or incomplete any inactive event, by use of the COMPLETION pseudovariable. This sets the appropriate bit in the event variable. The completion status may be inspected by means of the COMPLETION built-in function. The PL/I language also allows the programmer to inspect and change the status of an event, by means of the STATUS built-in function and pseudovariable.

WAIT Statement

The WAIT statement is implemented by a call to the resident library routine IBMDJWT. This is passed a set of parameters consisting of the addresses of the event variables and the number of events that have to be completed. If the number of events that have to be completed is not specified, all the events in the list must be completed.

The WAIT makes use of the DOS data-management WAITM and WAITF macro instructions. However, because of the differences between the facilities offered by the DOS system and the PL/I language, considerable housekeeping problems are involved for waits on more than one event. For waits on single events, the problems are small and are described at the end of this section.

When a WAIT or associated macro instruction is issued to the DOS supervisor, the event is considered to be

complete when input/output transmission is finished. In PL/I, however, a WAIT statement is not considered complete until any error-handling activity caused by the operation which was being waited on is finished. The error handling may include entry into an on-unit, and further WAIT statements may be executed in the on-unit. This process can continue to any number of levels of interrupt.

PL/I also allows the programmer direct control over the completion of an event by use of the COMPLETION pseudo-variable. Consequently, the PL/I programmer need not associate an event variable with an input/output operation, but can use it instead as a flag, setting the event complete at any point in the program.

WAIT or associated macro instructions issued to the supervisor are completed by setting a completion bit in the ECB (event control block) which is held in the DTF. At the PL/I level, completion is indicated by setting the completion bit in the event variable. Thus a WAIT operation is carried on at two levels, the PL/I level and the system level.

Housekeeping Problems

The problems involved in implementing the WAIT statement may be illustated with examples from the skeleton program in figure 11.6. Four problems arise. They are:

Problem 1: If an event being waited on in a multiple WAIT statement is completed in an on-unit entered while processing one of the other events in the statement, this must be made known to the first WAIT statement. Setting the event variable complete is not sufficient, because the event variable may be used again during the on-unit. Suppose that the RECORD condition is raised during the execution of the WAIT statement numbered 3 in figure 11.6, for the operation associated with event E1. The following then takes place:

1. Control passes to procedure M.
2. The statement WAIT(E2) is then encountered, and the program waits until event E2 is completed. When this occurs, the event variable is set complete and inactive.
3. Event E2 is then used in a further I/O operation (statement 11), causing the event variable to be set active and incomplete.

On return to the main program, there would be no way of determining from the event variable for E2 that the original event E2 had been completed. The problem is solved by the use of control blocks called event tables (EVTABs). An EVTAB is set up by the wait module each time a WAIT statement is encountered; it contains entries for each incomplete event specified in the statement. The entries are termed EVTAB elements. Each element is chained to its corresponding event variable and contains a bit that can be set to indicate that the event has been completed. In the above example, therefore, EVTAB elements for E1 and E2 are set up when the wait module is called at statement 3. When the on-unit is entered, the WAIT statement 9 causes a further EVTAB to be set up with an entry for E2. The event variable pointer is reset to address the latest EVTAB elements, and a field in this element is set to point to the previous EVTAB element for E2. When event E2 is completed (without causing any I/O conditions to be raised), the event variable and each EVTAB element for E2 is set complete and inactive, and a bit in the event variable is set to indicate that the chain of EVTAB elements is no longer associated with the event variable. When statement 11 is executed, the event variable is set active and incomplete. After the on-unit has been executed, the wait module sets the EVTAB element and event variable for E1 complete and inactive. It then tests any remaining EVTAB elements to determine whether they were set complete during an on-unit; in this case, it finds that the next EVTAB element (for E2) has been set complete and that there are no more events to process. Execution therefore continues until statement 4 is executed, at which time a new EVTAB element is created for E2 and chained to its event variable.

Problem 2: A method must be provided to signal that an event waited on in an on-unit is already being waited on in the procedure that caused entry to the on-unit. Suppose that the RECORD condition is encountered in the operation associated with E2 (statement number 2) during processing of the WAIT at statement number 3. The following then takes place:

1. Control passes to procedure M.
2. A further WAIT on E2 is encountered (statement number 9). Since E2 cannot now be completed, a mechanism must be available to raise the ERROR condition; otherwise, the program would never get out of the wait state,

The problem is solved by setting a flag in the event variable whenever an on-unit is entered during WAIT statement

processing. If the wait module is subsequently reentered from an on-unit, to process a WAIT on the same event, it finds that this bit is set and raises the ERROR condition.

Problem 3: If there is a GOTO out of an on-unit, this involves setting an event variable complete, and terminating the WAIT statement. Suppose the TRANSMIT condition is raised during the WAIT statement numbered 3, 4, or 9. The procedure L is entered and the following takes place:

1. E3, which is a dummy event, is set complete.
2. A GOTO is executed to the label BOOTLE.

If no other action were taken, the event that caused entry to the on-unit (either E1 or E2) would not be set complete; any subsequent WAIT on that event would thus cause the wait module to be invoked, with unpredictable results. The problem is solved by setting a flag bit in the current DSA whenever the wait module is called. (The method is similar to that used to cater for a GOTO out of a SORT exit, and uses the same flag bit.) If the GOTO module finds that the bit is set, it returns to the wait module; the wait module sets the event variable complete and inactive and then returns to the GOTO module to continue the GOTO out of the on-unit. Only the event that caused entry to the on-unit is set complete. Any other incomplete events specified in the WAIT statement are left incomplete.

Problem 4: If control reaches label BOOTLE without the TRANSMIT or RECORD condition having been raised, the event E3 can never be completed. Some method must be available of making this fact known, otherwise the program would go into an indefinite wait on an event that could never be completed. This problem is solved by setting an event variable active only when it is associated with an operation. Thus, if a WAIT statement specifies an event that is inactive and incomplete, the wait module causes the program to be terminated. (If a WAIT statement specifies more than one event and one of the events is inactive and incomplete, the program is not terminated immediately because it is possible, although unlikely, that the incomplete event will be completed by the COMPLETION pseudovariable in an on-unit entered as a result of an I/O condition raised while processing one of the other events specified in the WAIT statement.)

Control Blocks

Four control blocks are involved in the implementation of the WAIT statement. These are shown in detail in appendix B.

1. Event variable. Used to hold all information about the event at a PL/I level. Fields indicate whether it is active or inactive; complete or incomplete; whether it is already being waited on at a previous interrupt level; the type of operation with which it is associated. Each event variable contains the address of its associated ECB or CCB and, if it associated with an I/O event, the address of the FCB for the file.
2. ECB (event control block). Used to hold information about the event at the system level. For I/O events, ECBs are part of the DTF. For DISPLAY events, the equivalent control block is known as a CCB (channel control block).
3. EVTAB (event table). Created for each entry to the WAIT module; comprises an element for every incomplete event that is to be waited on. The EVTAB is held in a VDA acquired by the WAIT module.
4. ECB list. This is a list of ECB addresses that is created in circumstances that are explained below. The ECB list is held in the VDA described above, and acts as an argument list for the WAITM macro instruction.

Multiple-Wait Module (IBMDJWT)

The actions of the multiple-wait module, IBMDJWT, are shown in the flowchart in figure 11.7, and are described in detail in the publication DOS PL/I Resident Library Program Logic.

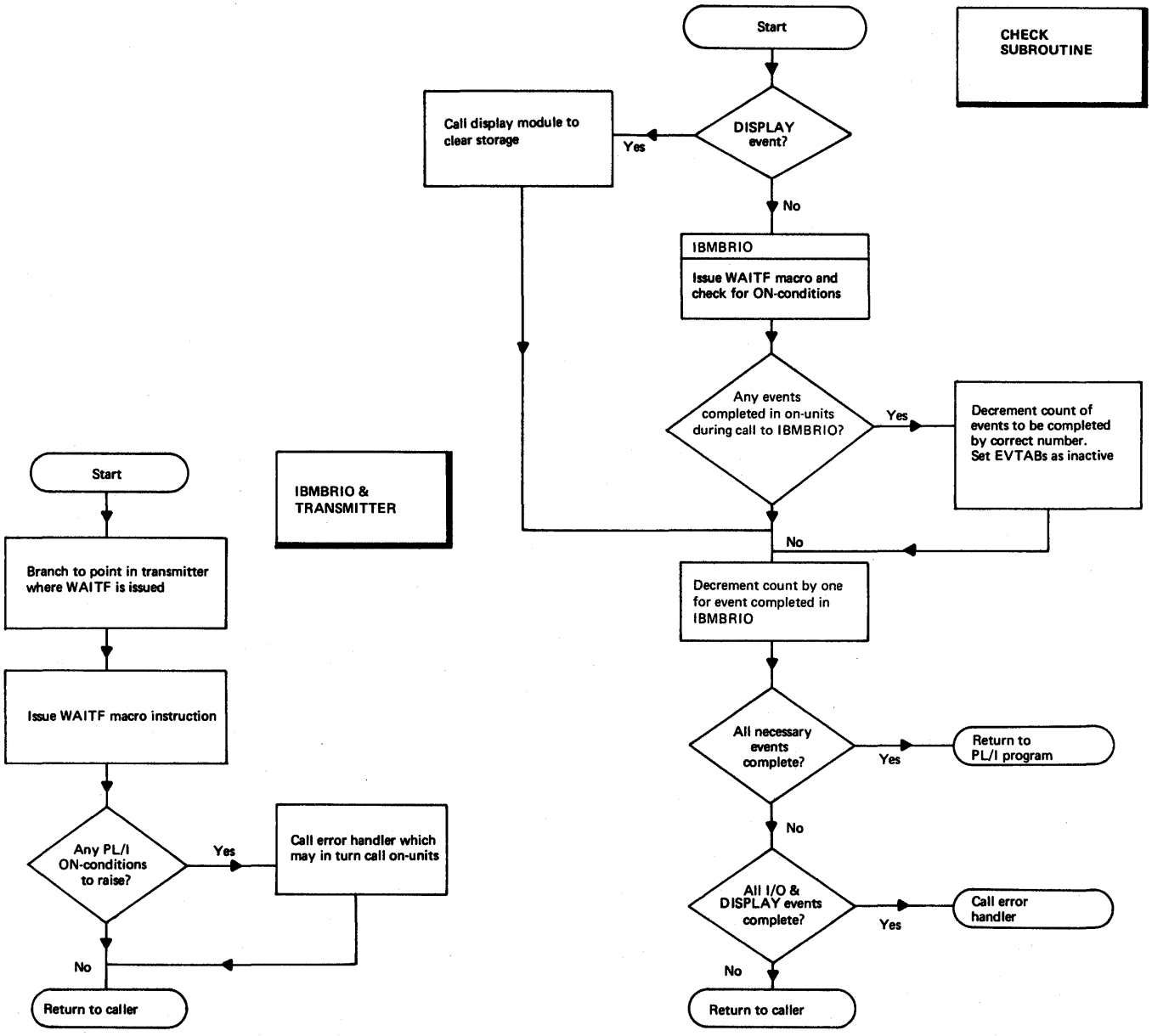


Figure 11.7. (Part 1 of 2). Simplified flowchart of modules used in execution of WAIT statement

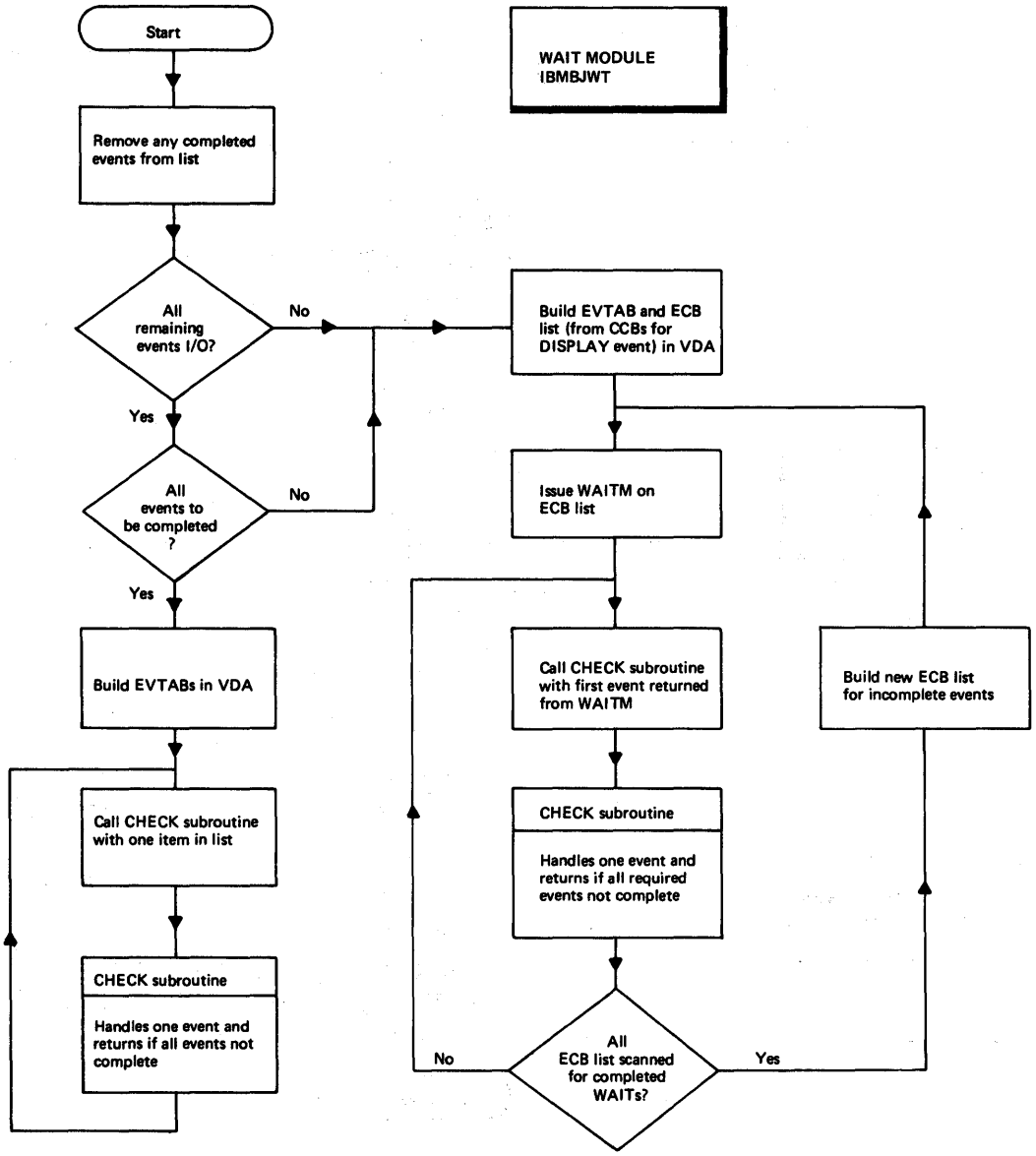


Figure 11.7. (Part 2 of 2). Simplified flowchart of modules used in execution of WAIT statement

As the flowchart shows, the WAIT module sometimes issues a WAITM macro instruction, and sometimes relies on the WAITF macro instructions in the PL/I transmitters. The reasons for this are as follows.

The WAITF macro instruction in the transmitter can only be used for I/O events, and only one transmitter can be called at a time. If only a certain number of the events in an event list need to be completed, it is uneconomic to pass these events one at a time to the transmitter, because the first event passed could be the last to finish. Consequently, whenever non-I/O events are involved and whenever only a specified number of events in an event list have to be completed, an ECB list is generated for all incomplete events and a WAITM macro instruction is issued.

The WAITM macro instruction returns control as soon as any event in the list is complete, thus allowing an event list to be handled efficiently when only a number of events have to be completed. For I/O events, it is still necessary to issue the WAITF macro instruction in the transmitter, even though the events are known to be complete. This is because the WAITF macro instruction carries out various checking functions.

Single-Wait Module(IBMGJWT)

When the WAIT statement is handled by the resident library routine IBMGJWT, only one event can be waited on in any WAIT statement. The housekeeping problems are therefore considerably less complicated than those encountered when handling waits on multiple events. No EVTABs, or ECB lists are needed. When the module is entered, it either calls IBMDRIO for an I/O event, or issues a WAIT macro instruction using the CCB for a DISPLAY event, calling the DISPLAY module IBMDJDS to clear the working storage and check for any transmission error.

Chapter 12: Debugging Using Dumps

The DOS PL/I Optimizing Compiler allows the programmer to obtain an execution time dump either by calling PLIDUMP or by specifying DUMP in the options statement. If he specifies DUMP in the options statement a dump will be given if the program is stopped because of the ERROR condition. In both these situations a formatted PL/I dump is given. A DOS system dump will not be produced except in exceptional conditions unless it is specified by the programmer using the Q option of PLIDUMP.

Certain types of program error, result in overwriting of the control information used by the PL/I error-handling routines, thus causing a program check to occur. If this occurs whilst a previous program check is being handled, a system dump will be generated even if NODUMP has been specified in the JCL. A dump is produced because the program check exit is reset during the handling of program check interrupts. (See chapter 7, "Error Handling" for further details.) If DUMP has been specified in the JCL, it is possible (though most unlikely) that a DOS system dump will be generated for other abnormal conditions, such as a rapid succession of program check interrupts.

Furthermore, it is always possible for the programmer to ask an operator to take a stand-alone dump at any point in the program. The need to do this should, however, occur only infrequently.

How to Use this Chapter

This chapter contains information on how to obtain and interpret dumps, and on how to identify compiled code, data, and control blocks. Some knowledge of the compiler's housekeeping scheme, described in other chapters of this book, is assumed. Trying to use a dump without this knowledge can result in a great deal of wasted time. To acquire a quick overall picture, chapter 1 and the introduction to chapters 6 and 7 should be read. A summary of how to use this chapter when debugging is given in figure 12.1.

This chapter is divided into three sections:

Section 1: How to obtain a PL/I dump

Section 2: Recommended debugging procedures

Section 3: Locating specific information

Section 1 explains how to obtain a hexadecimal dump of a PL/I program. It also gives some suggestions on the use of various compiler and PL/I options that may prove useful when debugging.

Section 2 offers two recommended courses for debugging a PL/I program by use of a dump. The first course deals with a PL/I dump that has been called from an ERROR on-unit and is being used to debug the problem program. The second course deals with the situation in which a DOS system dump has been generated, probably because the housekeeping control blocks have somehow been overwritten.

Section 3 describes how to find various data areas and other information. It is indexed and numbered for quick reference.

Before taking a dump, Section 1 should be read, because the methods used are not those familiar to programmers using the DOS system. Sections 2 and 3 are for use when debugging. Programmers who know what they are looking for should refer directly to the contents table in section 3. This will direct them to numbered sections which give details of how to find particular items. Programmers wishing to follow some organized plan can follow the recommended procedures in section 2. Section 2 crossrefers to the items in section 3, so that the details of the steps involved may be quickly found.

Section 1: How to Obtain a PL/I Dump

In order to get a formatted PL/I dump, the programmer can either include a call to PLIDUMP in his program, or specify the option DUMP in his JCL. If he specifies DUMP in his JCL a dump will be given if the program terminates with the essay.

The statement CALL PLIDUMP may appear wherever a CALL statement may legitimately be used. It has the following form:

```
CALL PLIDUMP
   (character-string-expression 1,
   character-string-expression 2);
```

HOW TO USE THIS CHAPTER WHEN DEBUGGING

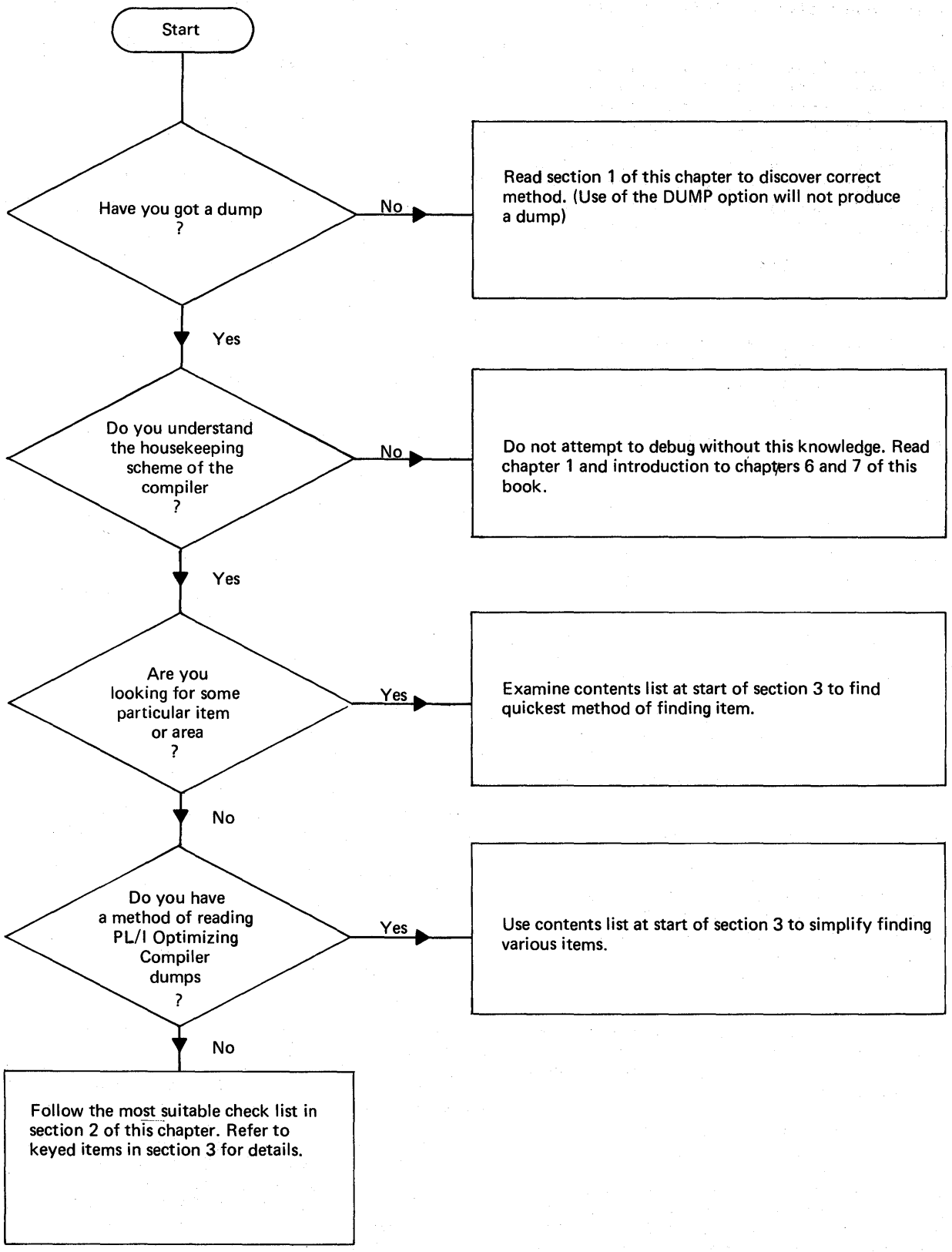


Figure 12.1. How to use this chapter when debugging

Character-string-expression 1 is a "dump options" character string consisting of one or more of the following dump option characters. The maximum length of this string is 256 characters. Defaults are underlined.

T Trace. A calling trace through all active DSAs is generated. When an on-unit DSA is encountered, the values of the relevant condition built-in functions are given. The reason for the entry to the on-unit is also given if the ERROR or FINISH conditions are raised as standard system action for another condition.

NT No trace. A calling trace is not given.

F File information. A complete set of attributes for all open files is given, plus the contents of all accessible buffers.

NF No file information required.

S Stop. The program will be terminated after the dump.

C Continue. Execution of the program will be continued after the dump.

H Hexadecimal. A hexadecimal dump of the partition will be given. If trace information is requested, the TCA and DSA addresses will be given. If file information is requested, the addresses of the FCBs will be given and the contents of all accessible buffers will be printed in hexadecimal notation as well as in character.

NH No hexadecimal dump required.

B Blocks. The contents of the TCA, TIA, DSAs, FCBs, and file buffers are printed in hexadecimal notation.

NB No block information required.

R Report. The lengths and addresses of the main areas of storage in use immediately before the call to PLIDUMP are given.

NR No report information required.

Q Quick dump. This gives a DOS system dump with none of the formatting and other information provided by PLIDUMP. The Q option only takes effect if all other options are negated. To obtain a DOS system dump using the Q option should be specified thus:

```
CALL PLIDUMP('Q NH ND NR NB NF NT');
```

There is no requirement for a dump identifier because this will not be reproduces on the DOS system dump.

NQ A DOS system dump is not required.

D Debug. Additional information about files will be given. This includes the name of the transmitter and the open module, and information on whether ENDFILE or an error has occurred on the file.

ND No debug. The additional files.

60 The hexadecimal notation will be translated into the 60 character set.

48 The hexadecimal notation will be translated into the 48 character set.

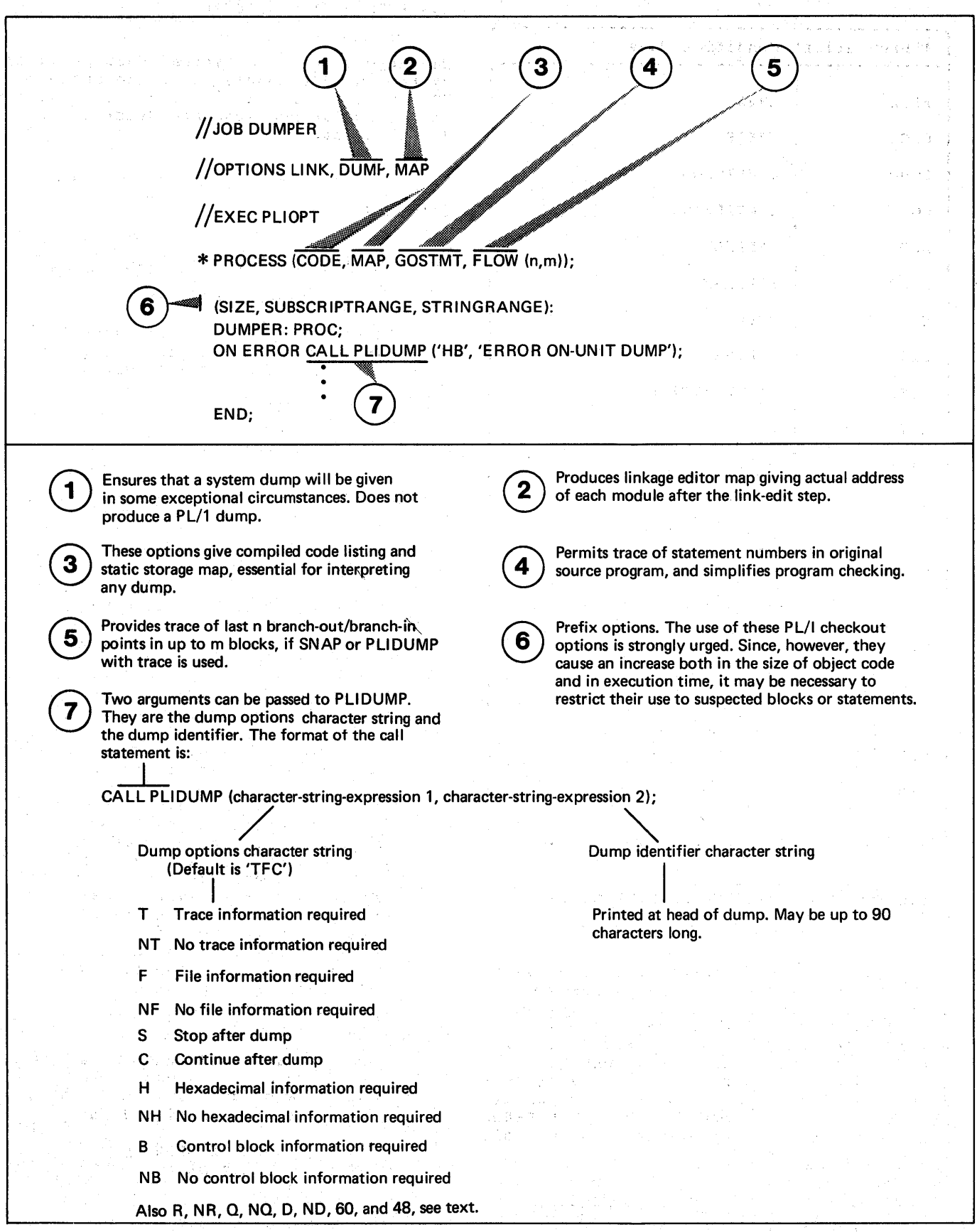
The default options are TFCRDNHNB60. That is, trace information, file information, debug file information, storage report, block information, no hexadecimal dump, continuation after the information has been put out, and translation into the 60 character set.

Options are read from left to right. Invalid options are ignored, and if contradictory options are coded, the rightmost options are taken. A further discussion of the output that results from each of these options is given later under the heading "Contents of a PL/I Dump".

Character-string-expression 2 is a "dump identifier" character string of up to 90 characters chosen by the PL/I programmer. It is printed at the head of the dump. If the character string is omitted, nothing is printed.

RECOMMENDED CODING

Since PL/I dumps are transmitted onto the standard file SYSLIST, it is important to insure that SYSLIST is assigned to a line printer device. PLIDUMP can be called from anywhere in a program, but the normal method used when debugging will be to call PLIDUMP from an on-unit. As continuation after the dump is one of the options available, PLIDUMP can be used as a snap dump to get a series of dumps of main storage throughout the running of the program.



(1) Ensures that a system dump will be given in some exceptional circumstances. Does not produce a PL/1 dump.

(2) Produces linkage editor map giving actual address of each module after the link-edit step.

(3) These options give compiled code listing and static storage map, essential for interpreting any dump.

(4) Permits trace of statement numbers in original source program, and simplifies program checking.

(5) Provides trace of last n branch-out/branch-in points in up to m blocks, if SNAP or PLIDUMP with trace is used.

(6) Prefix options. The use of these PL/I checkout options is strongly urged. Since, however, they cause an increase both in the size of object code and in execution time, it may be necessary to restrict their use to suspected blocks or statements.

(7) Two arguments can be passed to PLIDUMP. They are the dump options character string and the dump identifier. The format of the call statement is:

CALL PLIDUMP (character-string-expression 1, character-string-expression 2);

Dump options character string (Default is 'TFC')

- T Trace information required
- NT No trace information required
- F File information required
- NF No file information required
- S Stop after dump
- C Continue after dump
- H Hexadecimal information required
- NH No hexadecimal information required
- B Control block information required
- NB No control block information required

Also R, NR, Q, NQ, D, ND, 60, and 48, see text.

Dump identifier character string

Printed at head of dump. May be up to 90 characters long.

Figure 12.2. Coding dump options

Abbreviation	Condition Name
AREA	AREA
CHCK	CHECK
COND	CONDITION
CONV	CONVERSION
ENDF	ENDFILE
ENDP	ENDPAGE
ERR	ERROR
FIN	FINISH
FOFL	FIXEDOVERFLOW
KEY	KEY
NAME	NAME
OFL	OVERFLOW
REC	RECORD
SIZE	SIZE
STRG	STRINGRANGE
STRZ	STRINGSIZE
SUBG	SUBSCRIPTRANGE
TMIT	TRANSMIT
UFL	UNDERFLOW
UNDF	UNDEFINEDFILE
ZDIV	ZERODIVIDE

Figure 12.3. Abbreviations for condition names used in PLIDUMP trace information

By including the statement CALL PLIDUMP ('HB','dump identifier'); in an ERROR on-unit or by including DUMP in the options card "Contents of a PL/I Dump", it is possible to obtain a hexadecimal dump, with control blocks identified and formatted, should an error occur. If an ERROR on-unit is being included in a program, care should be taken that there are no further ON ERROR statements which might override the on-unit requesting a dump.

Suggested code for use when debugging with a dump is given in figure 12.2.

CONTENTS OF A PL/I DUMP

The appearance of a typical dump produced by the PLIDUMP modules with the options TFHB is shown in figure 12.4. The contents of particular sections are described in detail below.

Headings

The dump is headed by the line

PL/I DUMP

This is followed by the user identifier, if any, given as the second character string in the argument list of PLIDUMP.

Trace Information

A request for trace information results in the following output:

1. A trace of every procedure, begin block, and on-unit that is active at the time of the call to PLIDUMP. For procedures, the procedure name and statement number from which the procedure was called are given. If the 'H' option is requested, the offset of the statement is also given as well as the entry point address and DSA address. Also, if the entry point used is not the main entry point and the statement number option is in use, the main entry name is given.

2. For on-units, the values of any relevant condition built-in functions are given. The type of on-unit is given and, where the cause of entry into the on-unit is not self-explanatory, the cause of entry is also given (e.g., if an ERROR on-unit was entered because of a conversion error, this fact is given in the trace information). The on-unit type is specified, using a three or four letter abbreviation. A list of these abbreviations is given in figure 12.3.

3. When a hexadecimal dump is requested, the entry point address of each active block is also given, together with the address of its associated DSA.

4. When the compiler FLOW option is in effect, the flow statement table is also given.

5. If a hexadecimal dump is requested,

```
                                        * * * PL/I DUMP * * *

    USER IDENTIFIER :   EXAMPLE OF PLIDUMP

                                        * * * CALLING TRACE * * *

  (TCA ADDRESS 009E08 )

PLIDUMP WAS CALLED FROM STATEMENT NUMBER 3 AT OFFSET 00009E FROM A ERR  TYPE ON-UNIT WITH ENTRY ADDRESS 0078FC
       (AND DSA ADDRESS 00A628 )


                                        ERROR DIAGNOSTICS
          PL/I CONDITION DETECTED: CONV
          ONCODE    =     612                 INTERRUPT CODE, SEE LANGUAGE REFERENCE MANUAL
          ONCHAR    =I                        CHARACTER CAUSING CONVERSION ERROR
          ONSOURCE =IF THIS DOES NOT RAISE CONVERSION NOTHING WILL
                                              STRING CAUSING CONVERSION ERROR

ADDRESS OF ERROR HANDLER'S SAVE AREA 00A438
          REGISTERS ON ENTRY TO ERROR HANDLER

     REGS 0-7   FF00A438      0000A430      0000A3B8      6E009BE6      0000A1E8      0000A310      0000A3C8      80007A1A
     REGS 8-15  00000001      0000A33D      00000000      000088E8      00009E08      FF00A3E0      4E009DDE      0000928A

                                        END OF ERROR DIAGNOSTICS

WHICH WAS CALLED FROM A LIBRARY MODULE WITH ENTRY ADDRESS 009BE0 (AND DSA ADDRESS 00A3E0 )
WHICH WAS CALLED FROM A LIBRARY MODULE WITH ENTRY ADDRESS 008960 (AND DSA ADDRESS 00A0D8 )
WHICH WAS CALLED FROM A LIBRARY MODULE WITH ENTRY ADDRESS 007B08 (AND DSA ADDRESS 00A368 )


WHICH WAS CALLED FROM STATEMENT NUMBER 5 AT OFFSET 0000AC FROM A PROCEDURE EXAMPLE WITH ENTRY ADDRESS 007840
       (AND DSA ADDRESS 00A250 )

                                        * * * END OF CALLING TRACE * * *

          TRACE OF PL/I CONTROL BLOCKS

          TASK COMMUNICATIONS AREA

 ADDR. OFFSET
009E08 00000  00000000 00000000 FF009E08 FF035DF8  00000000 00009E18 00000000 0000A000  ..............]8..............
009E28 00020  00000000 00000000 00009F28 00000000  0000A038 00000000 00000000 00000000  ................................
009E48 00040  0000A020 00000000 00009996 00000000  00000000 80000000 80000000 00009000  ................................
009E68 00060  0000900A 00000000 00000000 00009988  0000998A 0000999A 0000928A 00000000  ................................
009E88 00080  582E0004 58EE0000 19DF478C 00B29500  C001478C 00AC180E 18E1181F 58FC00E0  ..................-.....  .....
009EA8 000A0  07FF0000 00000000 00000000 18DF9834  D0209160 D001078E 9140D001 478C00CC  ..................-.....  .....
009EC8 000C0  D203004C D0509120 D001078E D201D056  D0549180 D054071E 181F58FC 00F407FF  K.............K.........4..
009EE8 000E0  00009102 07FE0000 00000000 00000000  00000000 00008FC8 00000000 00000000  ..........................H.........
009F08 00100  00000000 00000000 00000000 00000000  00000000 00000000 00000000 00000000  ................................

          TCA IMPLEMENTATION APPENDAGE

 ADDR. OFFSET
009F28 00000  00036000 00000000 00000000 00000000  0000A080 000091C4 00000000 05F058F0  ..-.....................0.....0.0
009F48 00020  F04E051F 00000000 00000C00 00000000  00000000 00000000 00000000 00000000  0+..............................
009F68 00040  00000000 00000000 00000000 00000000  00000000 00000000 00000000 00000000  ................................
009F88 00060  00000000 00000000 00000000 00009260  00000000 00000000 000004A8 0000A248  ..............-................

          DYNAMIC SAVE AREA (LIBRARY)


                                        * * * PL/I DUMP * * *

CONTENTS OF REGISTER SAVE AREA
     REGS 0->7   FF00A750      00007A3C      5E007952      4E008F2E      0000A1E8      00007A3C      0000A2F8      FF00A750
     REGS 8->15  0000A6D8      0000A6F8      0000A250      00009F28      47103288      FF00A700      5E008F8C      FF00A778

 ADDR. OFFSET
00A700 00000  88004780 0000A628 00045866 5E008F8C  FF00A778 FF00A750 00007A3C 5E007952  ..............;.................;...
00A720 00020  4E008F2E 0000A1E8 00007A3C 0000A2F8  FF00A750 0000A6D8 0000A6F8 0000A250  +.......Y........8.......Q...8....
00A740 00040  00009F28 47103288 FF00A4C0 FF008080  F5F7F3F6 60D3D4F5 40C3D6D7 E8D9C9C7  .................5736-LM5 COPYRIG
00A760 00060  C8E340C9 C2D440C3 D6D9D748 40F1F9F7  F1400000 000078F0 90EBD00C 05301851  HT IBM CORP. 1971 .....0........
00A780 00080  41000138 5810D04C 1E015500 C00C47D0  301A58F0 C07405EF 58F0D048 90F01048  .....................0.....0...0..
00A7A0 000A0  50D01004 18D19288 D0009200 D0019107  D04F4780 30544100 00045810 D04C1E01  .....J.....................K.....
00A7C0 000C0  5500C00C 47D03050 58F0C048 05EF5000  D04CD703 D05CD05C 9240D064 D276D065  .........0.........P..*.*. ..K...
00A7E0 000E0  D06492D0 D05C5840 C0289608 D001503D  002050CD 00445000 D12C4510 308805F0  .....*. ..................J.....0
00A800 00100  58F0F052 051FD207 D0DC1000 4100D0DC  411030E2 5010D134 4110D0E4 9210D05E  .00...K.............S..J....U...;
00A820 00120  0A104110 33245010 D1305050 D0605820  C04058F2 000412FF 478030E2 91802000  ........J....-... .2......S....
00A840 00140  471030E2 4110D064 5010D050 4110D050  D703D054 D0549248 D0554120 D05405EF  ...S.............P..............
00A860 00160  41103138 5010D134 9280D05E 9120400C  47803104 58250000 D200D05C 200047F0  ........J....;..  .........K..*...0
00A880 00180  31381255 47803138 41000320 5810D04C  1E015500 C00C47D0 312258F0 C04805EF  ..............................0....
00A8A0 001A0  5000D04C 18714110 340A1807 0A0418F1  05EF5070 D04C4110 31AA9210 D05E5010  .............1..............;...
00A8C0 001CC  D134D214 D0953457 45E032C4 91805000  471031AA 9140D05F 471031AA 58650004  J.K.........0.......... ..........
00A8E0 001E0  58560000 91806006 4710317A 48560004  58660000 47F03188 D201D060 50004165  ......-.....-.....0...K.-.....
00A900 00200  00024850 D0601255 47D031AA 49503442  47D0319A 48503442 D210D069 346C0650  .....-.....K.....K.....K.......
00A920 00220  44503444 45E03318 41103224 5010D134  9240D05E 9180D05C 478031EE 41000B88  ..................J.. .;...*.........
00A940 00240  1A0C5810 D04C1E01 5500C00C 47D031D8  58F0C048 05EF5000 D04C1871 41103412  .....................Q.0.............
00A960 00260  18070A04 18F105EF 5070D04C 9140D05C  47803224 4100D8B8 5810D04C 1E015500  .....1........ .*.....K.....K.....
00A980 00280  C00C47D0 320E58F0 C04805EF 5000D04C  18714110 341A1807 0A0418F1 05EF5070  .....0.....K.....1....0.....1....
00A9A0 002A0  D04C4110 324C5010 D1349210 DC5E45E0  3318D20C D092344A D20ED09F 345D45E0  .....J....4....K....K....]....
00A9C0 002C0  33189120 D05C4780 32824110 32829220  D05E5010 D1344100 06405810 D04C1E01  .....*.....4....J.....  ......
00A9E0 002E0  5500C00C 47D03270 58F0C048 05EF5000  D04C1871 41103422 18070A04 18F105EF  .........0.....1....0.....1....
00AA00 00300  5810C028 4100101C 41101024 0A104110  34820700 501032A2 4110342A 450032A6  ................................
00AA20 00320  00000001 0A021BFF 9110D05C 471032B6  41F00008 58D0D004 58E0D00C 9808D014  ............*.....0.............
00AA40 00340  07FE4110 34820700 501032DA 41103432  450032DE 00000304 0000AC00 0A024120  ................................
00AA60 00360  D0634301 002C5021 002C4201 002C4301  00785021 00784201 00784301 00704122  ................................
00AA80 00380  00015021 00704201 00704100 D0589208  D05D186E 181D58F0 D05805EF  .....0...0.....]....0.....0....
00AAA0 003A0  07F658D0 10489580 D05E4770 F04692D0  D05C9621 D05C9640 D05F59D0 103C4770  .6.....:.....:..0....*....*. ......
00AAC0 003C0  F02ED202 10051051 0A11D203 D00C105D  D2331008 D01450D0 103CD207 1040D00C  0.K.....K.....K.........K.. ..
00AAE0 003E0  D2021005 10410A11 9540D05E 4770F062  9621D05C 9680D05F 59D0103C 4770F02E  K.........  .:..0....*..........0.
00AB00 00400  47F0F020 59D0103C 4780F020 9520D05E  4770F02E 4110F116 0A0290E5 D00C0540  .00.....0.....:..0...1....V....
00AB20 00420  18214850 20569188 205D4780 40309108  205D4710 4024947F 205D92F0 20630650  .........).. ....).. ....]..0....
00AB40 00440  47F04030 92F12063 4150003D 9680205D  06504110 40E058F1 001045EF 000C9240  .0 ..1.........]....  ..1.......
00AB60 00460  2063D277 20642063 12554720 4058D214  20964085 47F04024 40502056 94802050  ..K.........  .K...  ..0 . ......)
00AB80 00480  98E5D00C 07FE358E C9C2D4C4 D2D7E3C1  C9C2D4C4 D2E3C3C1 C9C2D4C4 D2C6C1C1  .V.......IBMDKPTAIBMDKTCAIBMDKFAA
00ABA0 004A0  C9C2D4C4 D2C4C4C1 5B5BC2C3 D3D6E2C5  5B5BC2D6 D7C5D5D9 5B5BC2C4 E4D4D740  IBMDKDDA$$BCLOSE$$BOPENR$$BDUMP
00ABC0 004C0  005AD200 D07D6000 5C405C40 5C4OC5D5  C440D6C6 405C405C 405C40D7 D361C940  ..K..'-.* * * END OF * * * PL/I
00ABE0 004E0  C4E4D4D7 405C405C 405CE4E2 C5D940C9  C4C5D5E3 C9C6C9C5 D9407A02 2B006006  DUMP * * *USER IDENTIFIER ....-.
00AC00 00500  00008000 0C000003 0000AC70 0000AC78  0800ACF0 3300D2D4 C4E3C640 40000000  ..............0...KMDTF  ...
00AC20 00520  00000000 00000800 002008F0 2400B113  80000000 00000000 00000000 0000FF00  ..........0.....................
00AC40 00540  00000000 00000000 13000000 00000000  00000079 47000000 0700AC3A 40000006  ..............................  ...
00AC60 00560  3100AC3C 40000005 0800AC60 20000001  D100B114 20000078 0500B113 60000079  .... ......-...............-...
00AC80 00580  3100AC3C 40000005 0800AC80 20000001  1E000540 30000081 00000000 00000000  .... ..... ..... ..... ..... .....
00ACA0 005A0  00000000 00000000 00000000 00000000  00000000 00000000 00000000 00000000  ................................
      LINE THE SAME AS ABOVE
00ACE0 005E0  00000000 00000000 47FF001C 47FF001C  0A320000 47F0F000 47F0F02A 47F0F02A  ....................00..00..00.
00AD00 00600  47F0F000 47F0F000 47F0F02E 47F0F186  C9D1D1C6 C3C2E9C4 F3F99026 F2585860  .00..00..00..01.IJJFCBZD39..2.-
00AD20 00620  100895FF 101E4780 F03E4A60 104A9103  1028A710 F1904740 F13A9140 102C4780  .........0.-........1.. 1.. ....
00AD40 00640  F05A948F 102C4530 F12C5846 00000640  9200F2A2 91F04000 4710F0A8 95C14000  0.......1..... ..2..0 ...0..A .
00AD60 00660  4780F0A8 95C24000 4780F0A8 95C34000  4780F0A8 4800F2A0 4130F279 41330012  ..0...B ...0..C ...0...2...2.....
00AD80 00680  D5004000 30004780 F0A80630 4600F090  9201F2A2 47F0F0D4 18304304 00004140  N. .....0.....0...2..00M........
00ADA0 006A0  00124334 F2791930 4780F0C0 4640F0B2  4940F270 47B0F0D0 920B6000 4530F12C  ....2.....0.. 0.. 2...0...-...1.
```

Figure 12.4. Example of PLIDUMP

the address of the TCA is printed at the head of the trace.

6. If either a hexadecimal dump or control block information has been requested, and any ERROR on-units are found, then the following information is also included:

   a. The address of IBMDERR's DSA.

   b. The contents of the general and floating point registers at the time IBMDERR was called.

   c. If there was an interrupt, the address of the interrupt.

   d. A trace of library DSAs back to the last compiled code DSA.

File Information

A request for file information results in the following output:

1. The default and declared attributes of all open files are given.

2. Buffer contents of all buffers are given. If a hexadecimal dump has been requested, the contents of the buffers are given in both hexadecimal and character notation. If no hexadecimal dump is requested, the contents are given in character notation only.

If the 'B' option is included the the contents of the FCB, ENVB and DTF, and for VSAM files the IOCB and ACB are given.

Due to the many possible variations in length of these blocks, the full control block may not always appear in this section of the PLIDUMP.

The description of the MEDIUM option contains the following abbreviations:

UNIT	Unit record device (card reader, printer etc.)
TAPE	Tape unit
DISK	Disk storage unit
INDE	Device independent

Debug Option

If the debug option is specified, additional information about files is provided. The name of the transmitter and open module associated with the file are provided, and other data regarding the status of the file is given. This includes whether an error has occurred on the file and whether ENDPAGE or ENDFILE have been raised.

Hexadecimal Dump

This is a dump of the partition associated with the program. The dump is set out in four columns. The first column contains the address in main storage. The second and third columns contain four fullwords each in hexadecimal notation. The fourth column is a reproduction of the second and third columns in character form. You can specify whether you want the translation to be into the 60 or 48 character set by using either the '60' or '48' option. 60 is the default.

The PL/I hexadecimal dump is headed by the contents of the communications region and, if no trace information was requested, by the values of registers 12 & 13 and the floating point registers on entry to the dump. It should be noted that if the dump was called from an on-unit, these values are not the values of the registers at the point of interrupt. The method of finding the register values at the point of interrupt is described in section 3, "Locating Specific Information."

Block Option

When the block option is used, the contents of the TCA, the TIA, and the DSAs in the LIFO stack (that is, all active DSAs) are printed in hexadecimal and character format. The absolute address is printed in the left hand column; the offsets within the block are then printed. This is followed by the contents of the block, first in hexadecimal and then in character notation. For DSAs, the type of DSA is shown; i.e., library DSA, procedure DSA, on-unit DSA, or dummy DSA. The contents of the FCBs, ENVBs, DTFs etc for any open files are printed in a similar format.

Report Information

The report option gives a report of the use of main storage immediately before the PLIDUMP was taken. It gives addresses and lengths of the major areas of storage and

shows how much storage in the partition is not in use. If the PL/I program was called from a program in another language that specified the storage area to be used, figures for the total used and unused storage may be inaccurate.

As described in chapter 1 and chapter 6, the partition used by a program compiled by the PL/I Optimizing Compiler is divided in a standard manner. A typical storage layout and related report are shown in figure 12.3. The partition is headed by the problem program which consists of compiler output link-edited with PL/I library routines. This is followed by an area of housekeeping control blocks known as the "program management area". The storage between the end of the program management area and the end of the partition is allocated dynamically, this area is known as the ISA (initial storage area). During execution, two storage stacks are created, one starting at each end of the ISA. One stack is the LIFO stack containing all storage that is acquired and freed on a last-in/first-out (LIFO) basis. This stack contains housekeeping information for each PL/I block, storage for automatic variables, and workspace. In the PLIDUMP storage report this area is referred to as "primary LIFO storage", and shown on the line numbered 04. The other stack starts at the end of the partition and contains all storage that is not acquired and freed on a last-in/first-out basis. This includes items such as controlled and based variables, transient library routines, I/O buffers, and control blocks associated with files. In the PLIDUMP storage report table this area is referred to as the "primary non-LIFO area" and shown on the line numbered 05. Both stacks extend into an area of unallocated storage known as the major free area. Within the non-LIFO stack certain areas may be freed which cannot be incorporated into the major free area. These are listed in the PLIDUMP storage report table with the line number 06 and headed "free area". In certain situations these free areas can be used for further segments of LIFO storage. Such segments are listed in the PLIDUMP report table with the line number 07 and are headed "LIFO overflow segment". When LIFO overflow segments have been allocated, the figures for total used storage and total unused storage on lines numbered 09 and 10 in the storage report table will be inaccurate. The total used storage will be overestimated. The overestimate will be smaller than the largest LIFO overflow segment.

Using the REPORT Option for Program Tuning

As well as its use for debugging, the report option can be used for estimating the optimum storage size for a program. When this is done PLIDUMP should be called when the maximum amount of storage is in use. This will be at the point when the greatest number of blocks are active, the greatest number of files are open, and the largest allocations of based or controlled variables have been made. PLIDUMP can be called in a number of places to get an accurate picture. As well as using the report option, it may be useful to use the trace and file options. The file option will tell what files are open, and the trace option will show the point in the program where the report was taken. These options are defaults, and to get a dump of this type the following PL/I statement should be included:

```
CALL PLIDUMP('ND','REPORT FOR TUNING');
```

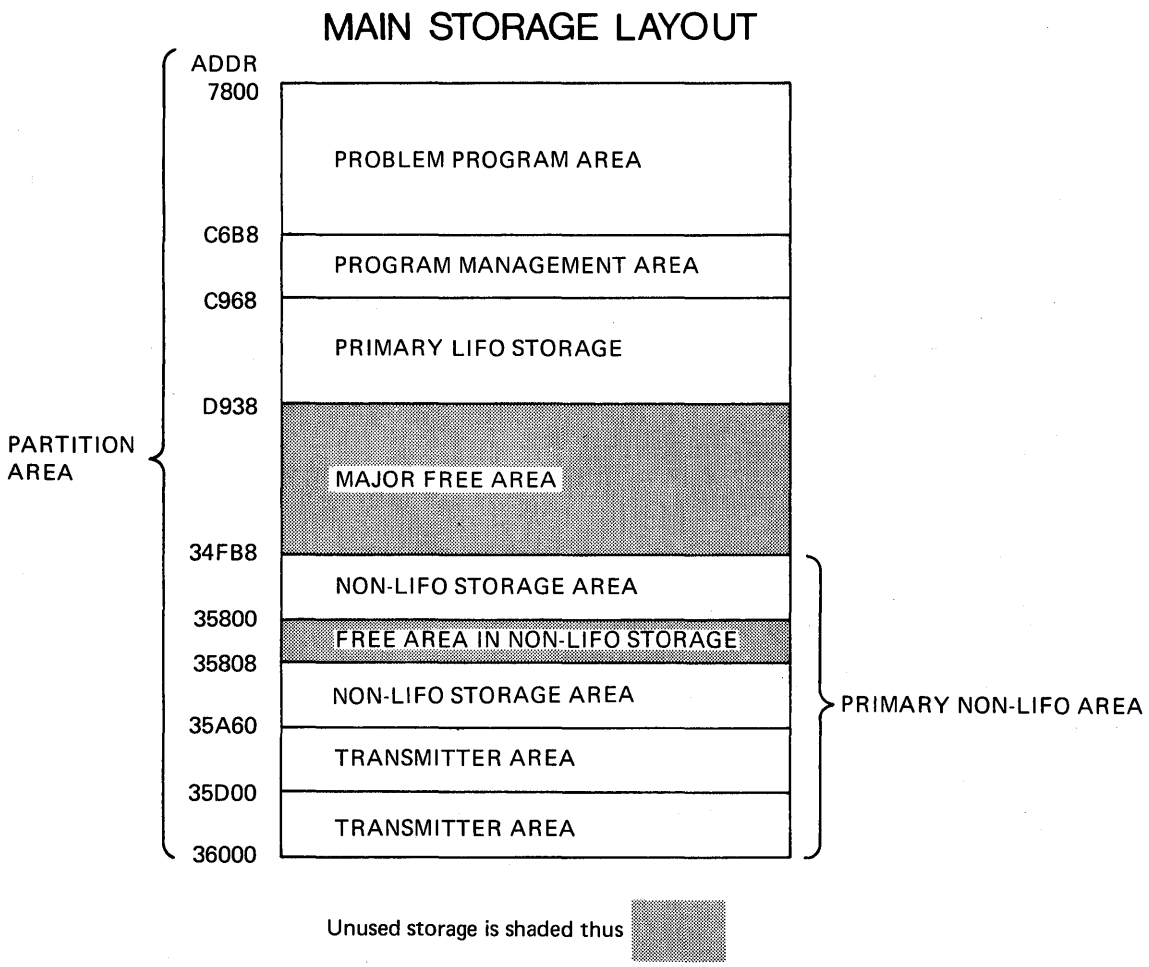
"ND" overrides the debug option and reduces the output. "Report for tuning" is the dump identifier and is used to show the purpose of the dump. When the maximum amount of storage used in the program has been established, the figure should be rounded ip to the nearest K bytes and a safety margin added. A suggested minimum is 2K bytes if SYSPRINT is open and 4K bytes if it is not. (The extra 2K for SYSPRINT allows error messages to be produced.)

Section 2: Recommended Debugging Procedures

The main difficulty in reading a dump of a PL/I program is knowing where to start. The signposts known to assembler language programmers are of little help. There are, however, five main sources of information to be considered when using a dump to debug a PL/I program. They are:

1. The statement number and the address where the error occurred (if the dump was taken after an error)

2. The type of error (if the dump was



ASSOCIATED STORAGE REPORT

```
                    * * * PL/I DUMP * * *
   USER IDENTIFIER : REPORT 1
                    * * * STORAGE REPORT * * *
01.PARTITION AREA          FROM  007800 TO  036000 LENGTH HEX  02F800  DEC  190464
02.PROBLEM PROGRAM AREA    FROM  007800 TO  00C6B8 LENGTH HEX  004EB8  DEC   20152
03.PROGRAM MANAGEMENT AREA FROM  00C6B8 TO  00C968 LENGTH HEX  0002B0  DEC     688
04.PRIMARY LIFO STORAGE    FROM  00C968 TO  00D938 LENGTH HEX  000FD0  DEC    4048
05.PRIMARY NON LIFO AREA   FROM  034FB8 TO  036000 LENGTH HEX  001048  DEC    4168
   INCLUDING
  06.FREE AREA             FROM  035800 TO  035808 LENGTH HEX  000008  DEC       8
  06T.FREE AREA TOTAL                              LENGTH HEX  000008  DEC       8
  08.TRANSMITTER AREA      FROM  035A60 TO  035D00 LENGTH HEX  0002A0  DEC     672
  08.TRANSMITTER AREA      FROM  035D00 TO  036000 LENGTH HEX  000300  DEC     768
  08T.TRANSMITTER AREA TOTAL                       LENGTH HEX  0005A0  DEC    1440
09.TOTAL USED STORAGE                              LENGTH HEX  007178  DEC   29048
10.TOTAL UNUSED STORAGE                            LENGTH HEX  027688  DEC  161416
                    * * * END OF REPORT * * *
```

Figure 12.5. A typical arrangement of main storage and an associated storage report.

taken after an error)

3. The values in the general registers when the dump was taken or when the error occurred

4. The chain of DSAs

5. The TCA

The first two of these items hold equivalent information to that held in the PSW in a DOS system dump. The last three items enable the housekeeping to be checked and the location of the control blocks and the program variables to be discovered. The methods of locating other information, given in section 3, refer to the key areas shown above.

When debugging, it is essential to have a listing of the object program and a linkage editor map. The object program listing allows the programmer to study the instructions that are being carried out and to find various control blocks in static storage. The linkage editor map allows the programmer to identify particular parts of the executable program phase and, for instance, to identify the routine associated with each DSA. It is also very desirable to have a variables offset map generated when the compiler MAP option is used.

THE CONTENTS OF A DUMP

The PLIDUMP and the DOS system dump both consist of a dump of the partition that is associated with the program. The principal contents of the partition are shown in appendix A. More detailed descriptions of the contents of main storage can be obtained from chapters 1 and 2. The partition contents will also appear in a stand-alone dump. The partition contains all information that is connected with the program. This will comprise the compiled code, any link-edited PL/I library modules, any transient PL/I library modules that are currently loaded, housekeeping control blocks, and all program variables.

DEBUGGING PROCEDURES

The best approach to a dump depends on the problem to be solved and must therefore be left largely in the hands of the programmer. However, two suggested courses of action are given in this section.

These courses cover two situations:

1. When PLIDUMP has been called from an ERROR or other on-unit

2. When a DOS system dump has been generated

Other possible situations are when a dump is taken at a specified point in the program, or when a stand-alone dump is taken. No attempt is made to suggest a course of action in these circumstances, because the reason for the dump being taken is not predictable. However, in such cases, the main storage situation can be investigated by following the methods itemized in section 3 of this chapter.

Throughout each of the two recommended procedures given in the following paragraphs, there are cross-references to the methods given in section 3. The cross-references consist of the keys by which the methods are identified; for example, H6, D5.

PL/I Dump Called from On-Unit

If a PL/I dump is called from an ERROR on-unit it can be assumed that the housekeeping system of the program is working. If it were not working, the dump would probably not have been generated.

A large amount of diagnostic information will be available at the head of the dump. An error message will have been generated, and this will provide a useful starting point. The first step should be to examine the error type and the point at which it occurred. ONCODE and other condition built-in function values should be examined, as should the trace information. A suggested procedure is the following:

1. Examine the error by means of the ONCODE and any other relevant built-in function values. These values are held in the trace information. (The meanings of oncodes are given in the language reference manual for this compiler.)

2. Find the location of error (P1) and in which block the error occurred (H12). If error occurred in library module, see H14.

3. Examine the trace to see if it appears as expected.

4. Examine the information in the file buffers, and check that file attributes are as expected. This information will be printed in the dump heading.

5. Check the values of any variables involved in the interrupt (V1-V6).

6. Check values of registers to see if dedicated registers are pointing to correct areas (H8 & H9). Distinguish between compiled code and library register usage.

7. Check housekeeping (H1-H16) starting with area most directly concerned with type of statement in which the error occurred.

8. Check values of all variables in the program (V1-V6).

9. Check logic of code being executed from object listing.

DOS System Dump

A DOS system dump consists of four columns of hexadecimal figures. The first column is the address in main storage; the second and third columns are the contents of main storage printed in hexadecimal notation; the fourth is the contents of main storage in character form, with a period for unprintable characters. Each column contains four fullwords. The dump is headed by the register values at the point when the dump was taken, and this is followed by the address of the communications region.

A DOS system dump is generated when there is a failure of the error-handling modules, or of the module that prints the PL/I hexadecimal dump, or when it is requested by the Q option of PLIDUMP or when there is not enough main storage to continue. It should be noted that the failure of these modules is more likely to be caused by the overwriting of essential information than by an error in the modules themselves.

A DOS system dump will not normally be produced for program checks, because a program check exit is set by the PL/I housekeeping routines, so that the system returns all program checks to the error handler. In the error handler itself, the program check exit is reset so that a program check interrupt will result in a dump.

Thus, a DOS system dump will be produced if the program check exit, which is normally set by the program initialization routines to prevent a dump, has been reset during the program, or, possibly, has not been set at all. The second alternative is extremely unlikely. A third possibility is that the program check exit itself is not working, and the STXIT macro in the initialization routines did not successfully set the program check exit. The most probable of these suggested causes is that the program check exit has been reset by the program. The program check exit is always reset for the duration of error handling or PLIDUMP, to prevent looping should an interrupt occur. (See chapter 7, "Error Handling.") If an interrupt occurs during error handling, a dump is therefore produced. An interrupt in the error-handling routines indicates either that the error-handling routines are at fault, or, more probably, that some of the control information of the error-handling routines has been overwritten during the execution of the program. The most practical solution may be to re-run the program with SUBSCRIPTRANGE, STRINGSIZE, and STRINGRANGE enabled.

However, having obtained a DOS system dump, the following debugging procedure may be adopted.

1. Determine whether dump was caused by program check. This can normally be discovered from the message printed on the page before the dump. If no message is printed, inspect the program interrupt key (PIK) in the communications region (D6).

2. Determine in which routines the error occurred. (D1 and 2 for address of interrupt, H2 for associating address with code.) Verify that this module is one called from error handler. (H3 and H10 for identifying module; figure 12.6 for modules called from error handler.)

3. Investigate the error that caused entry into the error handler. This can be done by examining the contents of IBMDERR's DSA (H7) and the associated ONCA (H6). See whether incorrect information passed to the error handler could be causing a failure. If the instruction is within the program control section shown on the linkage editor map the address can be associated with a statement (see H.2).

4. Locate instruction causing interrupt. This is done by looking for the PSW in the partition save area (D0).

5. Inspect this instruction to see if it appears to have been overwritten, bearing in mind the cause of the interrupt, e.g.,

   a. is it a valid instruction?

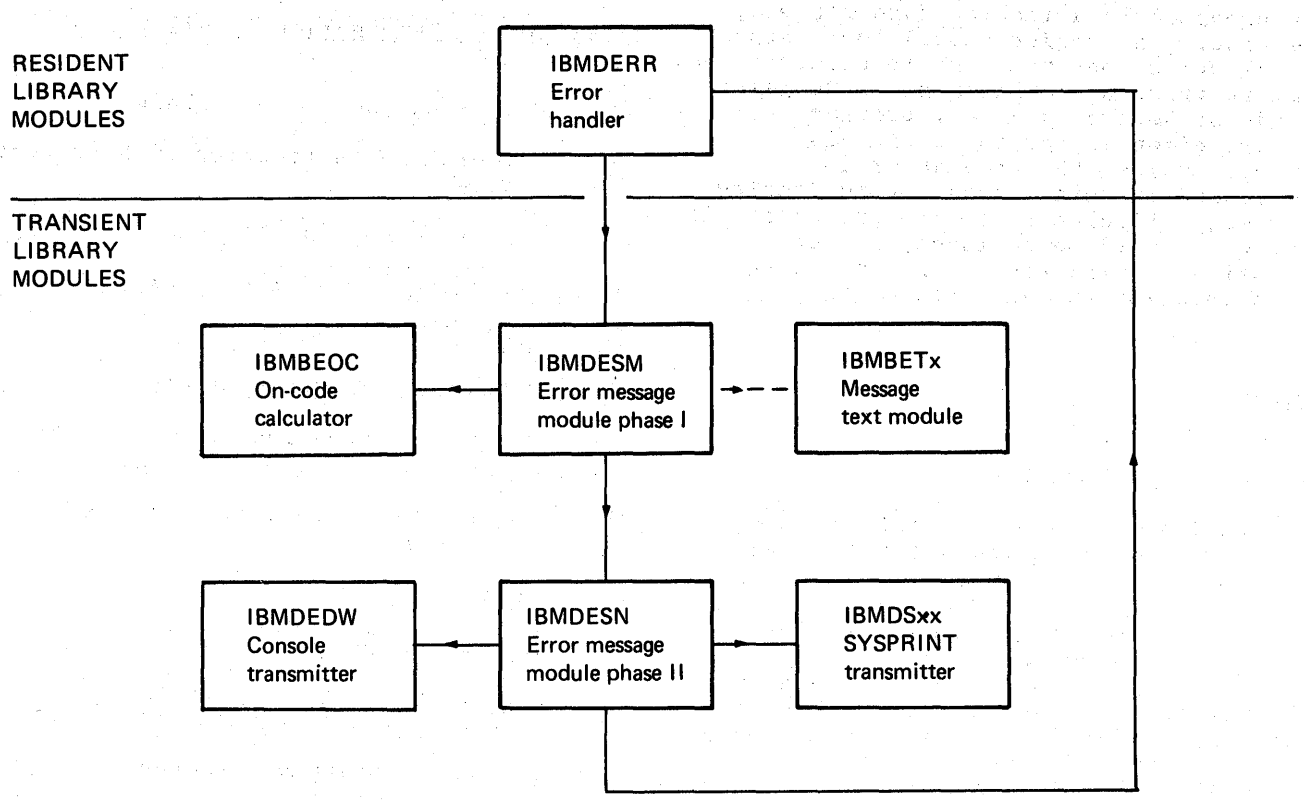


Figure 12.6. Error message group of modules

b. is it a branch to a protected address?

6. Inspect the TCA(D5) to ensure that all error-handling addresses are correct.

7. Investigate the housekeeping fields, starting with the DSA chain (H1-H3), then the chain of ONCAs (H5,H6).

8. If none of the above actions produces any results, an error in the error-handling modules must be presumed. This cannot be investigated without a listing of the modules. Meanwhile the cause of the original entry to the error handler has been discovered and can possibly be avoided by altering the source program so that the error does not occur. The trouble should nonetheless be reported, because a bug in the PL/I error-handling routines has apparently been discovered. It must be emphasized that the cause of entry into the PL/I error handler was not the cause of the system dump.

9. If the interrupt is not in the error handler or PLIDUMP, or one of the routines they call, the highest probability is still that the program check exit was altered in the error handler and that an invalid branch has been made from one of the addresses in the TCA. A careful check should therefore be made in the TCA. (See appendix B for map of TCA.) If this fails to produce results, return to stage 2 of the above procedure.

It may be possible to use the program by avoiding the cause of entry into the error handler discovered in 3 above. However, as the error is probably due to some kind of overwriting, simply bypassing the statement identified in 3 may not have the desired results.

Section 3: Locating Specific Information

This section tells the reader how to discover information from the dump. It has been produced in a modular form for easy reference. The reader should look through the following contents list to discover the items in which he is interested. Suggested methods of debugging a PL/I program from a dump are given in section 2 of this chapter. Unless the programmer is experienced in using dumps, or is looking for some particular item, the procedures in section 2 should be followed, rather than attempting to find various items through the information in this section.

CONTENTS

Control Blocks and Fields

C1 Quick guide to identifying control fields

KEY AREAS OF A PL/I DUMP

P1: Statement Number and Address where Error Occurred (Dump Called from On-Unit only)

Information required is the point at which the condition that caused entry to the on-unit occurred. This is identified in the trace information. If no trace information is generated, the method suggested for DOS system dumps can be employed. If the condition occurred in compiled code, the machine instruction being executed can be identified on the object program listing. This is done by subtracting the address of the program control section from the address of the interrupt and looking at this offset in the object program listing. The instruction thus found will be the one after the instruction that was last executed.

Alternatively the statement number table can be used (see H2).

P2: Type of Error (Dump Called from On-Unit only)

The type of error is identified in the trace information, in terms of the type of on-unit entered and the reason for entry. The on-code is also given, thus providing further indication of the cause of the condition. If the dump was called from an ERROR on-unit, an error message should have been generated before the dump. This again will give the cause of the error.

If no trace information has been generated, the type of error can be discovered from the error code appearing in the ONCA associated with the interrupt. The method for finding the ONCA is described in H5.

P3: Register Contents at Time of Error or Dump Invocation

If trace information has been generated, the contents of the registers must be found from the save area in the DSA. The addresses of all DSAs appear in the trace information. The register contents required will depend on the situation. If PLIDUMP was called from an on-unit, the register contents at the time the condition was raised will be most useful, unless the condition was raised in a library module. If the condition was raised in a library module, the contents of the registers at the point where the library call was made will probably prove more useful.

The method of finding the register contents is as follows:

1. Find the DSA of IBMDERR. (For release 4.0 and subsequent releases of the compiler, this DSA contains X'EEEE' in the second and third bytes of the save area.) The value of register 13 will be found in the chainback field at offset 4 of this DSA. The first byte will contain the segment no. (probably 'FF') and can be ignored for addressing purposes.

2. If the interrupt was a program check interrupt (see figure 12.7), the contents of registers 14 and 15 will also be stored in the DSA, register 14 at offset '5C'(92) and register 15 at offset '60'(96) from the head of the DSA.

3. Registers 0 through 11 will be stored in the save area of the previous DSA, starting at offset '14'(20).

4. If the interrupt was a software interrupt, the registers will be stored at offset 'C'(12) of the DSA before IBMDERR's DSA in the order 14 through 11. See figure 12.7.

Discovering if interrupt was program check interrupt: If trace information is available, a check can be made on whether IBMBERRA or IBMBERRB was called. IBMBERRA is entered after program check interupts, IBMBERRB after software interrupts. If no trace information is available, the simplest method of discovering if the interrupt was a program check interrupt is to inspect bit 7 in byte X'56' (86) in IBMDERR's DSA. This is set to zero for program check interrupts, and to 1 for other interrupts.

Finding registers if interrupt occurred in library routine: If on-unit was entered from a library module, a search back through the DSA chain to the first compiled code DSA should be made. This can be discovered from the trace information or by following the backchain from IBMDERR's DSA (offset 4 in each DSA) until a procedure block, begin block, or on-unit DSA is found. This may be determined from flag

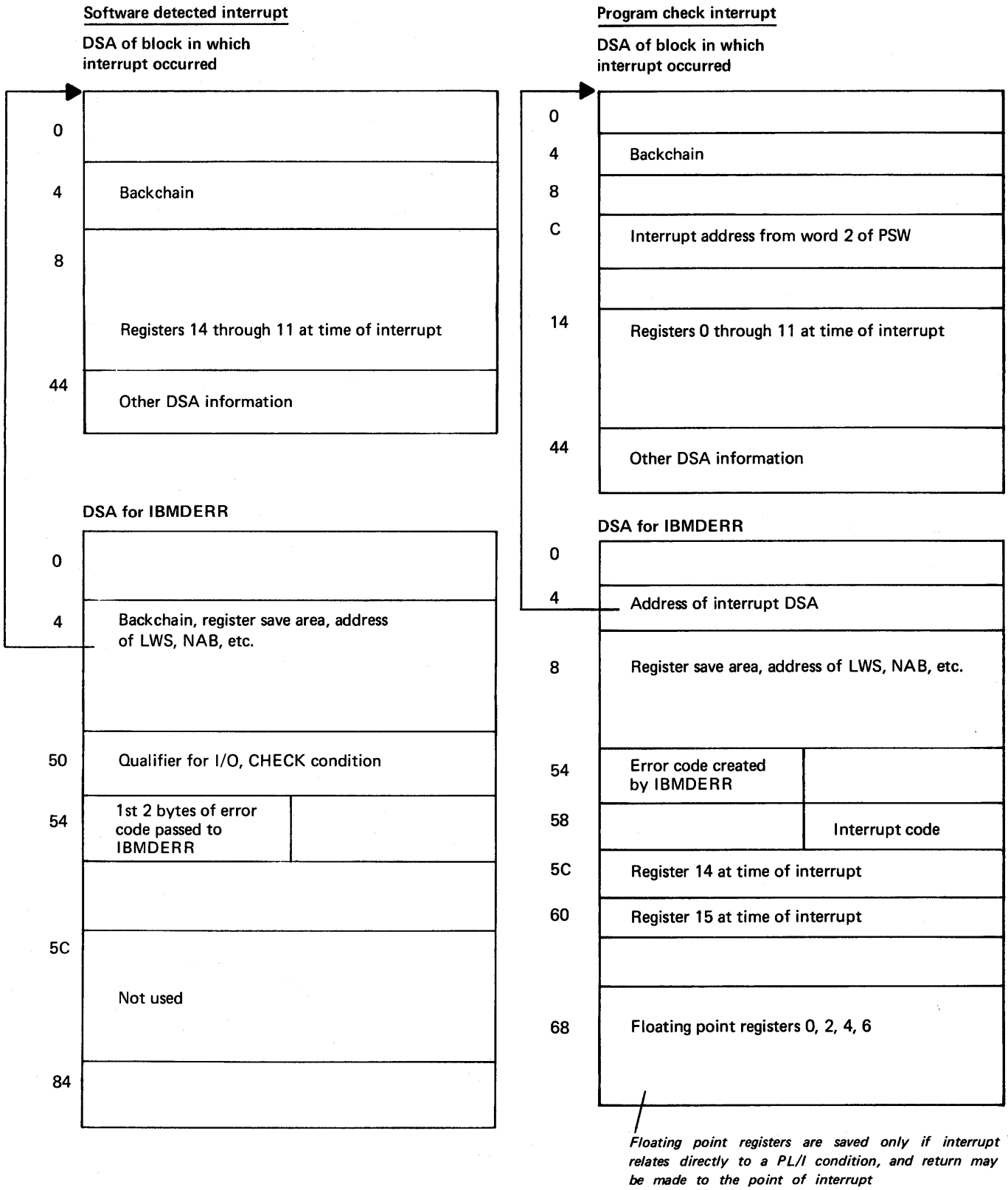


Figure 12.7. Information stored by IBMDERR after a program check and a software interrupt

BYTE 1	PL/I CONDITION IF ANY	BASE NO.
X'02'	ZERODIVIDE	320
X'03'	FIXEDOVERFLOW	310
X'04'	SIZE	340
X'05'	CONVERSION	600
X'06'	OVERFLOW	300
X'07'	UNDERFLOW	330
X'08'	STRINGSIZE	150
X'09'	STRINGRANGE	350
X'0A'	SUBSCRIPTRANGE	520
X'0B'	AREA	360
X'0C'	ERROR	009
X'0D'	FINISH	004
X'0E'	CHECK	510
X'0F'	CONDITION	500
X'10'	KEY	050
X'11'	RECORD	020
X'12'	UNDEFINEDFILE	080
X'13'	ENDFILE	070
X'14'	TRANSMIT	040
X'15'	NAME	010
X'16'	ENDPAGE	090
X'CD'		9250
X'CF'		1000
X'D3'		9200
X'D5'		3500
X'D7'		4050
X'D9'		5050
X'DF'		5000
X'E1'		9050
X'E3'		1000
X'E5'		4000
X'E7'		no on-code*
X'E9'		4050
X'EB'		0003
X'ED'		1000
X'EF'		1550
X'F1'		1500
X'F3'		2000
X'F5'		3768
X'F7'		3000
X'F9'		3800
X'FB'		3900
X'FD'		9000
X'FF'		8090

* Permanent WAIT. Generates message and terminates.

Figure 12.8. Error code field lookup table

bits 4 and 5 of a DSA, as follows:

Bit 4	Bit 5	DSA
0	0	Procedure block DSA
0	1	Begin block DSA
1	0	Library DSA
1	1	On-unit DSA

The value of register 12 can only be discovered in a DSA prior to a compiled code DSA, as it is not stored by the library when entering a routine. This means that the dummy DSA always contains the value of register 12. Register 12 should point to the TCA, whose address is also given at the head of trace information.

No trace information generated: If no trace information has been generated, the register values on taking the dump will be printed at its head. The address of the DSA for PLIDUMP will be in register 13. The chainback can then be followed to find the DSA for IBMDERR. The DSA for IBMDERR can be recognized if an on-unit is involved, because it will be the DSA before the on-unit DSA. IBMDERR's DSA will always be headed by a flag of hexadecimal '88' meaning that it is a library DSA in LIFO storage. To identify IBMDERR's DSA for certain, register 15 of the previous block's DSA must be inspected to see if it points to the module IBMDERR. The position of IBMDERR is shown in the linkage editor map.

P4: The DSA Chain

The addresses of the DSAs are given in a PL/I dump if trace information and a hexadecimal dump are requested. If trace information is not requested, the address of the DSA for the dump routine can be obtained from register 13 at the head of the dump. The chainback field is held in the second word of the DSA. When the dummy DSA is reached, this chainback field will be set to zero. The DSA chain passes through DSAs in LIFO storage and DSAs in LWS.

See H1 and figure 12.10 for details of how to follow the DSA chain.

P5: The TCA

The address of the TCA is given in a PL/I dump. If 'B' (block option) is specified in the dump-options character string, the complete TCA (including the appendage) is printed separately from the body of the dump.

KEY AREAS OF A DOS SYSTEM DUMP

The method of finding the key areas of a DOS system dump depends on finding the partition save area. The partition save area contains the old PSW and the register values at the point of interrupt; from these the key items can be identified. The format of the partition save area is shown in figure 12.9.

D0: Partition Save Area

The partition save area immediately precedes compiled code and is found in the following manner.

1. Use the linkage editor map to find the absolute address of the start of the program control section.
2. Look immediately before the control section in the body of the dump. This will be the start of the partition save area. The partition save area is 120 bytes long and usually starts with the characters 'NO NAME'. Normally it starts at the head of background storage, which is headed by the letters ---BG---.

The contents of the registers and the old PSW are located in the partition save area at the offsets shown in figure 12.9.

D1: Address of Interrupt

The address of the interrupt can be found from the second word of the PSW, which gives the address of the instruction following the point of interrupt. To find the associated statement number see H2.

Finding the statement number is not likely to prove useful because of the circumstances in which a DOS system dump is generated. The address found will usually be the address at which the error handler was entered before the program check exit was altered. The reason for entry into the error handler is not the cause of the dump.

	Partition save area
0	Normally 'NO NAME'
4	P.S.W.
C	Register 9
10	Register 10
14	Register 11
18	Register 12
1C	Register 13
20	Register 14
24	Register 15
28	Register 0
2C	Register 1
30	Register 2
34	Register 3
38	Register 4
3C	Register 5
40	Register 6
44	Register 7
48	Register 8
4C	Length reserved label area
50	Partition start time
54	Floating point register 0
5C	Floating point register 2
64	Floating point register 4
6C	Floating point register 6

Figure 12.9. Partition save area

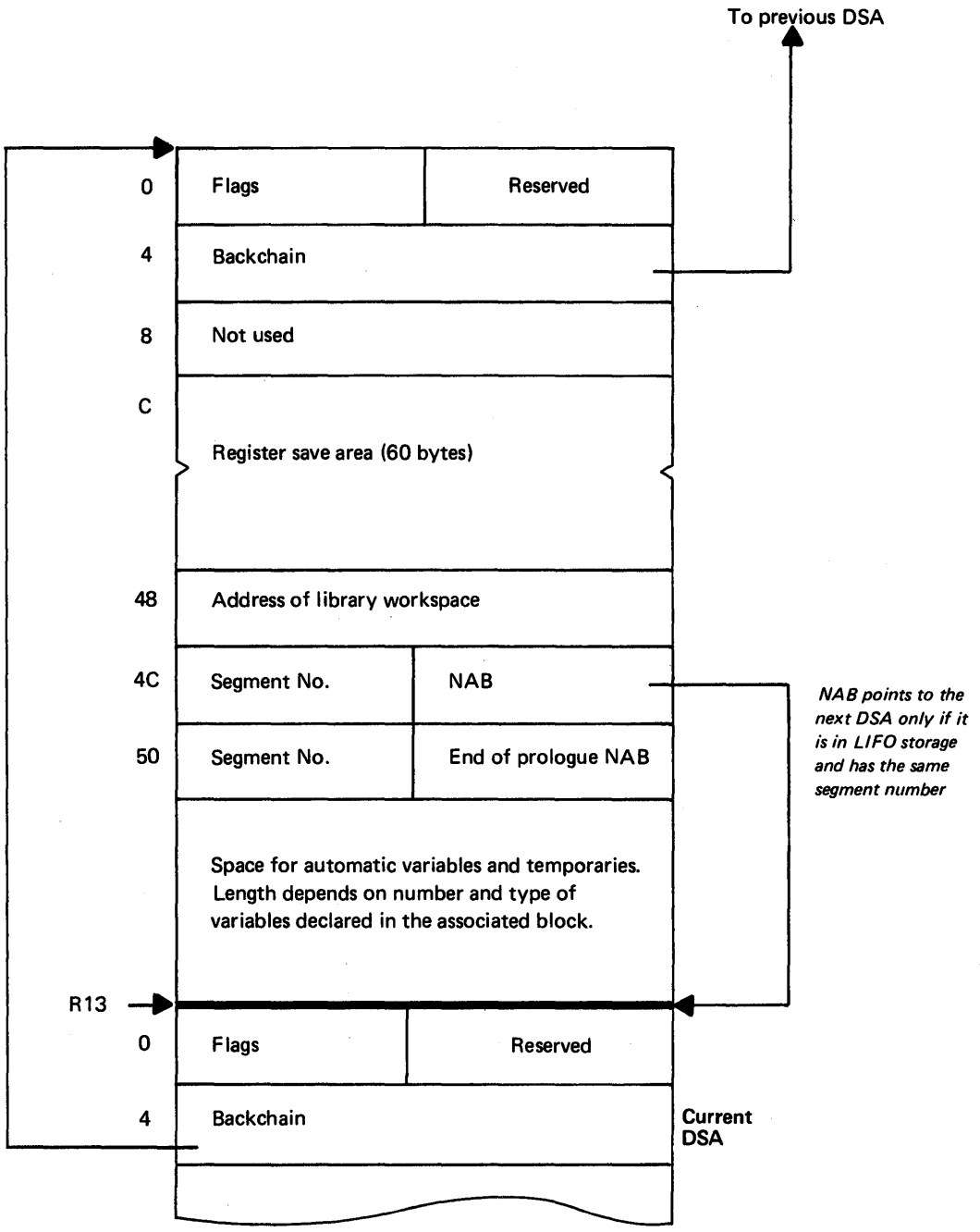


Figure 12.10. DSA chaining

D2: Type of Interrupt

The type of interrupt can be found from the first word of the PSW (see Principles of Operation for details).

D3: Register Contents at the Point of Interrupt

These are printed at the head of the dump, or can be found in the partition save area.

D4: The DSA Chain

Register 13 should point at the most recent DSA. The back chain can be followed from offset '4' of each DSA. See figure 12.10.

D5: The TCA

Register 12 should point at the TCA.

D6: Finding Program Interrupt Key (PIK)

The program interrupt key is held at offset hexadecimal '2E' (46) from the start of the communications region. It stores information on where the interrupt occurred; from this information it is possible to decide which PSW to inspect for the address of the interrupt.

Task in which interrupt occurred	PIK value	Relevant PSW
Background	X'10'	Old program PSW
Foreground 2	X'20'	Old program PSW
Foreground 1	X'30'	Old program PSW
I/O	X'50'	Old I/O PSW
Supervisor	X'60'	Old supervisor PSW

D7: Finding the Communications Region

The address of the communications region is printed at the head of a DOS system dump.

STAND-ALONE DUMPS

S1: Finding Key Areas in Stand-Alone Dumps

From the linkage editor map, the address of the end of the executable program phase can be identified. If the program is a straightforward PL/I program, the TCA will start at the first doubleword boundary following the end of the executable program phase. If PL/I has been called from assembler, COBOL, or FORTRAN, the address of the TCA may have been specified by that language. In either case, the TCA is readily discovered. From the TCA the dummy DSA can be found. The flag byte of the dummy DSA is set tc X'82'. The dummy DSA can be located frcm a field at offset X'10' (16) in the implementation-defined appendage. (The address of the implementation-defined appendage is held at offset X'28' (40) from the start of the TCA.) The main procedure DSA can then be located and further information found by following through the DSA chain. NAB (offset X'4C' decimal 76) in the dummy DSA always points to the main procedure DSA.

HOUSEKEEPING INFORMATION IN ALL DUMPS

H1: Following the DSA Backchain

Eacn DSA holds a backchain address in the second word. This word holds the address of the previous DSA. The end of the chain is marked by the dummy DSA whose first word contains the flag hexadecimal '82'. The backchain in the dummy DSA points to the external save area or is zero if the program was called from the system. (See P4 or D4 for finding the DSA chain).

H2: Associating Instruction with Correct Statement and Program Block

Statement Number and Program Block
The statement number and entry point associated with the interrupt will normally be given in a PLIDUMP. However, if they have to be found by the programmer, he should follow the method used by the error message modules.

Statement number: It must first be established whether the GOSTMT option is in effect. This will be indicated in the listing for the compilation. If the listing is not available it will be flagged in the compiled code DSA. (Flag bit 13 of the DSA flags is set to '1'B.) If this bit is not set the table of offsets and statement numbers may be available, ii this is not available statement numbers and offsets must be deduced from the object program listing. The method of using the table of offsets is described below under the heading "Using the Table of Offsets". If both statement numbers and the table of offsets are available it will probably be faster to use the table of offsets rather than the statement number table.

The statement number is found by use of the DSA chain as described below:

1. Find the chain of DSAs. The most recent DSA should be addressed by register 13.

2. If the DSA found is not a compiled code DSA, (flag bits 4 and 5 set to '00'B, '01'B or '11'B) the interrupt was not in compiled code. If the interrupt was in compiled code, the interrupt address can be directly asscciated with a statement number.

   If the interrupt was not in compiled code, the address at which compiled code was left must be discovered and this address associated with a statement number. To find the address at which compiled code was left:

   a. Chain back along the DSA chain until a compiled code DSA is reached (flag bits 4 and 5 set to '00', '01', or '11'B).

   b. The register 14 address saved in the DSA (offset 12 X'C') will be the point to which the library module or other module would have returned if the call had been successfully completed.

The address thus found is the address to be associated with a statement number.

3. Chain back one DSA to the DSA before the compiled code DSA that has been discovered in 1 or 2 above. The register 15 value in this DSA (offset 16 X'10') is the entry point of the block. If this appears to give an invalid result, check to see whether the DSA is one of those used in interlanguage communication (flag bit 7 set to '1'B and bit 0 of flags 2 (offset X'76') set to '1'B). If this is the case chain back one more DSA and try again.

4. At offset 8 from the entry point of the block, the address of the statement number table will be held.

5. Calculate the offset between the value in the first word of the statement number table and the address for which a statement number is required. If the address for which a statement number is required is less than the address in the first word of the statement number table, then either an invalid branch has been made, or a compiler generated subroutine is being executed. If it is possible that a compiler generated subroutine is being executed return to the compiled code DSA and attempt to find a statement number associated with the values held first in register 6, and, if this gives an invalid or improbable result, then in register 14. If the second word in the statement number table is less than the offset between the address for which a statement number is required and the first word of the statement number table, it is not within the program control section and an erroneous branch has been made out of the program.

6. If the offset is more than X'7FFF' the statement number will be held in the second or subsequent sections of the table. Obtain the number given by translating the offset into binary and ignoring the last 15 bits and step down this number of sections of the table. (For example, if the offset was X'8FFF', translate to binary = '1000 1111 1111 1111'B, ignore last 15 binary digits =1, therefore step down one section of the table. If the offset was X'18FFF' the binary would be '0001 1000 1111 1111 1111'B. Ignoring the 15 right hand bits leaves '11'B therefore step down three sections of the table.)

   The address of the second section of the table is held at offset X'8' in the table, the address of the third section is held at the head of the second section, the address of the fourth section at the head of the second section and so forth.

7. When the correct section of the table has been identified, search for the first offset in the table that is greater than or equal to the offset that is being searched for. The statement number is in two-byte hexadecimal format and immediately precedes this offset.

Procedure name: To find the entry point name, a chainback is made beyond the first procedure DSA found on the chain. Register 15 in the save area before this procedure DSA will point to the entry point of the procedure. (Procedure DSA have flag bits 4 and 5 set to '00'B. The register 15 value is held at offset 16 X'10'.)

The entry is preceded by a one byte field that holds the number of characters in the name. This one byte field is in turn preceded by the entry point name.

Using the table of offsets: Statement numbers can also be found by comparing them with the offsets in the offset and statement number table generated by the compiler when the OFFSET option is

specified.

Offsets are held from each primary entry point or a procedure or on-unit. To use the table of offsets find the entry point used by the program in the manner described above. Find the primary entry point for the procedure. (If the primary entry point was not used look at the object program listing to see the relationship between the entry point used and the primary entry point.) Note, the offsets given are from the point marked *REAL ENTRY in the object program listing. This point is one byte after the end of the primary entry point name.

If the interrupt occurred in an on-unit it may be necessary to discover the type of on-unit entered before it can be identified. This is done by inspecting the DSA before the DSA of the on-unit. This DSA will be for IBMDERR. At offset 84 (X'54') in this DSA the first byte of the error code will be held. Compare this with the values in figure 12.8. This will give an associated PL/I condition. It will be the on-unit for this condition that has been entered. If there is more than one on-unit for the condition, the on-unit entered must be deduced by studying the dump, and source and object listings. If the register 15 value appears to be invalid this may be caused by rechaining in interlanguage processing (see chapter 13). If this is possible, chain back one more DSA and try again. (To check if this has occurred see 3, above under "Statement Numbers").
Compare address of instruction with linkage editor map. This will give the name of the control section for compiled code or a library module. If the address is not included in the linkage editor map, the address is probably in a transient routine, unless an invalid branch has been made. (See H11.)

H3: Following Calling Trace

The calling trace can be followed because branches within the program are always made on registers 14 and 15. Hence register 15 in each DSA save area points to the address that was branched to from that block. Register 14 points to the address to which control passed when the block was completed. By comparing these values with the linkage editor map, it is possible to associate each DSA with the correct module of code. By following the backchain of DSAs (H1) it is possible to do this for every DSA and so discover the calling trace. The calling trace is printed in a PL/I dump.

H4: Associating DSA with Block

DSAs are associated with code by finding the register values in the register save area (H7) and using the fact that all branches are made via registers 14 and 15. Register 14 in any DSA points to the instruction after the point at which control left that block. Register 15 points to the address at which the next block was entered. By comparing these addresses with the linkage editor map, the DSA can be associated with the correct block of code.

H5: Finding Relevant ONCA

When an interrupt has occurred in the error handler and a system dump has been produced, it is possible to discover the information that the error handler would have used to generate appropriate error messages. The ONCA holds values for the condition built-in functions. The appropriate ONCA can be found in the following manner.

1. Find the DSA before that of IBMDERR (follow back the DSA chain until register 15 in the save area points to IBMDERR). See H1, H3, H7. If this is a library DSA in library workspace (flag bit 4 set to '1'B and flag bits 0 and 5 set to '0'B) continue to step 3.

2. Find the LWS addressed from this DSA. This is held at offset X'48' (72).

3. Find the offset from the LWS to the ONCA. This is held at offset 2 in the LWS.

4. Add the offset to the address of the DSA in LWS.

Interpreting the Error Code: The first two bytes of the error code are held at offset 4 in the ONCA. These two bytes normally be translated into an oncode which refers to the type of interrupt. The meaning of the oncode can be found in the language reference manual for this compiler. For PL/I conditions the first byte indicates the PL/I condition that has occurred. (See figure 12.8).

To translate the first two bytes of the error code into the oncode:

1. Find the base number associated with the value in the first byte (offset X'54'). Figure 12.8 is a table of the byte values and their associated base

number. (Base values are in decimal).

2. Take the right hand five bits of the second byte (offset X'55') and translate these into decimal.

3. Add this value to the base number found in 1 above. The result is the oncode for the interrupt that caused entry into the error handler.

Example:

Error code X'1266'

Look up base value = 80 (equivalent PL/I condition UNDEFINEDFILE)

Translate second byte to binary X'66'=0110 0110 binary

5 right hand bits =0 0110=6 decimal

Oncode=6 +base value=86

H6: Following the Chain of ONCAs

ONCAs are used to hold condition built-in function values. They are chained together, one being provided for every level of interrupt. The chainback field is in the first word of the ONCA. The dummy ONCA is marked by a chainback field of zero.

H7: Finding Information from IBMDERR's DSA

The information held in IBMDERR's DSA is that which is used by the error message modules for information about the error. It can be useful if the messages have not been generated, because the information can be deduced from the DSA. The contents of IBMDERR's DSA are shown in figures 12.7. See H4 for associating DSAs with correct code.

Interrupt Address: The address of the interrupt that caused entry into the error handler is held at offset 12 (X'C') in the DSA preceding the error handler's DSA. To find the statement number of the interrupt see H2.

H8: Finding and Interpreting Register Save Areas

Register save areas are held at offset X'C'(12) in all DSAs, including DSAs in LWS. Offsets and registers are shown in figure 12.11. Each DSA holds the register values as they were on exit from its block.

Note: Library routines store at least registers 14 through 4, and up to registers 14 through 11; compiled code routines store registers 14 through 12. Thus the address of register 12 can always be found in the dummy DSA although it may not be in other DSAs. The contents of the register save area in the DSA of the block that called IBMDERR are slightly different from normal if the interrupt was a program check interrupt. See figure 12.7.

H9: Register Usage

Register usage is fully discussed in chapter 2, "Compiler Output." A summary of register usage, showing which registers are always used for a particular purpose, is given in figure 12.12.

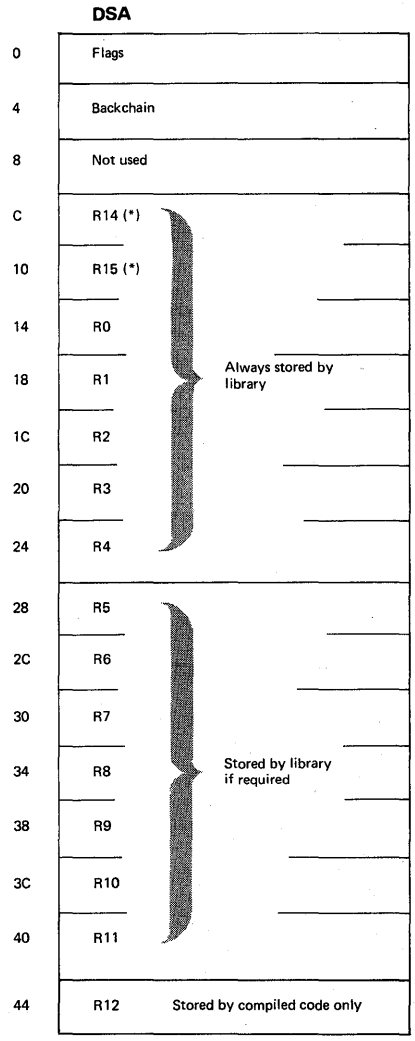


Figure 12.11. The register save area in the DSA

H10: Following Free-Area Chain

The free-area chain connects the areas of non-LIFO dynamic storage that have been used and freed, but have not been absorbed into the major free area. See chapter 6, "Storage Management." The chain starts at offset 8 in the implementation-defined appendage, which is addressed from offset X'28'(40) in the TCA. The end of the chain is marked with a zero entry. The length of each item is held at offset 0. The address of the next free area at offset 4. If there are no further free areas the word at offset 4 is set to zero.

Register	Compiled code usage	Library usage
R0	Work register	Work register
R1	Work register	Work register
R2	Program base (*,**)	Work register
R3	Static base (**)	Program base (**)
R4	Work register (Temporary base if DSA>3896 bytes)	Work register
R5	Work register	Work register (if used)
R6	Work register	Work register (if used)
R7	Work register	Work register (if used)
R8	Work register	Work register (if used)
R9	Work register	Work register (if used)
R10	Work register	Work register (if used)
R11	Work register	Work register (if used)
R12	TCA pointer (**)	TCA pointer (**)
R13	Current DSA pointer (**)	Current DSA pointer (**)
R14	Branch register	Branch register
R15	Link register	Link register

(*) The contents of the program base register are saved during in-line record I/O and TRT instructions

(**) Dedicated register, i.e., the contents remain unchanged throughout the execution of the associated compiled code or library routine

Figure 12.12. Register usage

H11: Action if Interrupt Occured at Address not in Linkage Editor Map

If the interrupt occurred at an address that is not mentioned in the linkage editor map, there are two possible explanations:

1. The interrupt occurred in a transient module.
2. An invalid branch has been made.

To test whether the error is in a transient module, the backchain should be followed to see whether register 14 in the previous DSA points to an area reasonably close to the point of interrupt. If so, an invalid branch can be discounted. Further chain-backs should be made along the DSA chain (H1) until a register save area is found that contains a register 14 which points to an area in the linkage editor map. This will be the routine that called the transient routine. It will usually be possible to deduce which transient routine is involved from the calling module and the context of the code.

If the most recent DSA is that of IBMDRIO the transient routine will be one associated with record I/O. The 5th, 6th, and 7th letters of the name of the module will be held in bytes 8-10 of the module. The letters are most simply found by inspecting the character translation of the dump. The first of these letters will always be R. The module will also be on a transmitter chain that starts at a word addressed from offset X'18'(24) in the TCA appendage, which is addressed from X'28'(40) in the TCA.

H12: Block Structure of Program (Static Backchain)

The block structure of the program can be followed from the address held at offset X'58'(88) in each compiled code DSA. This address holds the address of the compiled code DSA of the statically enccmpassing block. The chain thus formed is known as the static backchain.

H13: Forward Chain in DSAs

The forward chain in DSAs is not supported by the compiler. However, a forward chain through the LIFO stack can normally be followed by use of the NAB pointer. The NAB pointer is held at offset X'4C'(76) from the head of each DSA. The last pointer in the chain points to the major free area. If the NAB pointer contains anything except 'FF' in its first byte, the chain cannot be followed, because it is not contained in a single LIFO segment. The address required is held in the last three bytes of NAB; the first byte contains the segment number (see C1). The forward chain includes only those DSAs in the LIFO stack and does not include any DSAs in LWS.

H14: Action if Error is in a Library Module

If the source of an error is traced to a library module, it may appear that there is little that the programmer can do unless he has a listing of the library module. However, the fact that the interrupt or the error was discovered during the execution of a library module does not mean that the library module itself was in error. Before such a conclusion can be drawn, a check must be made on the data that is being passed to the module.

H15: Discovering Contents of Parameter Lists

Parameters are passed in a list of words pointed to by register 1, except during stream I/O. To find the position of a parameter passed to a program, find the value of register 1 in the save area of the DSA (see H4) of the calling block. Register 1 will then locate the parameter list. This can be compared with the static storage listing. The name of the called routine can be discovered (H3). The correct parameters are given in the appropriate library PLM.

H16: Finding Main Procedure DSA

The main procedure DSA can be found by following the backchain of DSAs to the dummy DSA. The address of the main procedure DSA will be given by the last 3 bytes of NAB in the dummy DSA. This is held at offset X'4C'(76) in the dummy DSA.

The address of the dummy DSA is held at offset X'10'(16) in the TCA appendage, which is addressed from offset X'28'(40) in the TCA. The dummy DSA can be recognized by the presence of X'82' in the flag byte.

FINDING VARIABLES

The value of the variables in the program at the point of interrupt can be discovered by using offset map and the compiled code listing as a guide to their addresses, and then finding these addresses in the dump. The method used depends on the type of variable.

V1: Automatic Variables

Automatic variables can be found by using an offset from the DSA of the block in which they were declared. This information appears in the variables offset map generated when the compiler MAP option is used. If the compiler MAP option has not been used, the information can be deduced from compiled code. (For finding the DSA associated with the block, see H4).

V2: Static Variables

Static variables are normally addressed by an offset from register 3. This offset is given in the variables offset map generated when the compiler MAP option is used. If the compiler MAP option has not been used, the offset can be deduced by studying the listing of compiled code. The value of register 3 can be found in the save area of the DSA. (For finding the DSA associated with the block, see H4).

V3: Controlled Variables

As described in chapter 2, internal controlled variables are addressed by an anchor word that is held in the static control section. This can be identified from compiled code, where it will normally be addressed by an offset from register 3. Typical code would be:

From this it can be deduced that the address of K is held at offset X'88' from register 3.

External controlled variables are

addressed from control sections that are shown in the linkage editor map. The variable starts at an offset of '8' from the address held in the control section. The first four bytes contain a pointer to previous allocations of the variable, or are zero if there are no previous allocations of the variable.

V4: Based Variables

Based variables are located by finding the value of the defining pointer. This value is found by using one of the methods described above to find static, automatic, or controlled variables. If the pointer is itself based, its defining pointer must be found and the chain followed until the correct value is found.

Typical code would be the following:

For X BASED (P), with P AUTOMATIC

```
58 60 D 088          L 6,P

58 E0 6 000          L 14,X
```

P is held at offset X'88' from register 13, and this address points at X.

Care must be taken when examining a based variable to ensure that the pointers are still valid.

V5: Area Variables

Area variables are located in one of the ways described above, according to their storage class.

Typical code would be:

For area variable A declared AUTOMATIC

```
41 60 D 088          LA 6,A
```

The area would start at offset X'88' from register 13.

V6: Variables in Areas

Variables in areas are found by locating the area and then using the offset to find the variable.

CONTROL BLOCKS AND FIELDS

For simplicity, the methods of finding various control blocks are placed in an alphabetic table. Details of the control blocks can be discovered from the relevant chapters (see index) or from appendix B.

As well as control blocks, various other items are included in the list. Where necessary, cross-reference is made to other sections in this chapter.

C1: Quick Guide to Identifying Control Fields

Automatic variables	see "Variables"
Backchain	
DSA backchain	offset X'4' in DSA
ONCA backchain	offset X'0' in ONCA
BOS	offset X'8' from TCA
Controlled variables	see "Variables"
DED	deduced from object program listing
Diagnostic statement table	addressed from offset '8' from entry point of main procedure
DFB	addressed from offset X'40'(64) in TCA
DSA	addressed by register 13 (see P3 and D3)
DTF	addressed from offset X'18' (24) in FCB
ENVB	addressed from offset X'14' (20) in FCB
EOS	offset X'C' (12) in TCA
Event variable	deduced from object program listing and knowledge of parameter lists of I/O and wait modules
FCB	identified in PL/I dumps. Open file statement listing.

Flow statement table	addressed from offset X'4C' (76) in TCA
Filename	addressed from offset X'10' (16) in FCB
Free-area chain	offset '8' in implementation-defined appendage, which is addressed from offset X'28' (40) in TCA
Locator/descriptor	deduced from object program listing
LWS	addressed from offset X'48' (72) in every DSA
Module name (when interrupt occurs in library module)	comparing address of error with link-edit map
NAB	offset X'4C' (76) in DSA
ONCA	the offset of the associated CNCA is held in a halfword at offset '2' in each section of LWS
CNCB - start of dynamic CNCB chain	offset X'60' (96) in DSA
- first static CNCB	offset X'5C' (92) in DSA
On-cells	addressed from offset X'70' (112) in DSA
OCB	deduced from object program listing and parameter list of open module, IBMDOCL
PSW	see D2
Parameter lists	object program listing and static storage map
Register values	See P3 and D3
RCB	object program listing and static storage map
Statement frequency count table	X '80' in the TCA
Static storage	addressed by register 3 in compiled code. See P3 and D3
SIOCB	object program listing
Symbol table	Static listing
Symbol table vector	Static listing
Start of program	linkage editor map
Segment number	first two bytes of BOS, EOS, or NAB. 'FF'=1,'FE'=2 etc.*
TCA	addressed by register 12. See P3 and D3
Variables	
automatic	offset (shown in variables offset map) from DSA of block in which they are declared. See V1
based	address of the pointer must be deduced from the object program listing. This gives the address of the variable. See V2
controlled	address referenced in compiled code holds latest allocation of the variable. A chain-back through the previous allocation can be made using the header chain. See chapter 2, and V3
static	deduced from offset from register 3 in variables offset map. See V4
area	as for other variables depending on storage class. See V5
Variables in areas	find address of area. Find variable from offset within areas shown in compiled code. See V6

*When the first two bytes of EOS and BOS are greater than NAB, it means that an extra segment of storage has been used, but not yet freed. See chapter 6, "Storage Management."

Chapter 13: Interlanguage Communications

The DOS PL/I Optimizing Compiler allows subroutines compiled on certain IBM COBOL or FORTRAN compilers to be used in PL/I programs compiled on the optimizing compiler. Similarly, it compiles PL/I programs that can be run as subroutines of either COBOL or FORTRAN programs.

Facilities are also provided to overcome the addressing problems that arise when passing arguments to assembler language routines. These facilities are described under the heading "Options Assembler" later in this chapter.

A full description of how the interlanguage communication facilities can be used is given in the language reference manual and the programmer's guide for this compiler. A detailed description of the library routines involved is given in the resident library PLM. This chapter explains the basic design principles used. It will assist in understanding the situation in main storage during the execution of a program involving interlanguage calls.

Background to Interlanguage Communication

The major problems involved in allowing procedures written in PL/I to be used with programs written in COBOL or FORTRAN are:

1. The existence of different data types in the different languages.
2. The different methods of holding data aggregates in the different languages.
3. PL/I's use of locators when passing areas, arrays, strings, and structures as arguments.
4. The need for programs compiled on PL/I and FORTRAN compilers to have a specially initialized environment in which to operate.

The first of these problems must be solved by the programmer himself, by ensuring that arguments passed between the routines are of suitable data types. (Information in the language reference manual for this compiler enables the programmer to do this.)

The other problems mentioned above are handled automatically by the interlanguage communication facilities of the compiler. These problems are summarized below.

DIFFERENCES IN DATA AGGREGATES

Structures in PL/I and COBOL, and arrays in PL/I and FORTRAN, are held in different manners.

COBOL structures are mapped as follows. Working from the start, each item is aligned to its required boundary in the order in which it is declared, the structure starting on a doubleword boundary.

PL/I structures are mapped by a method that minimizes the unused bytes in the structure. Basically, the method used is first to align items in pairs, moving the item with the lesser alignment requirement as close as possible to the item with the greater alignment requirement. The method is described in full in the lahguage reference manual.

Take, for example, a structure consisting of a single character and a fullword fixed binary item. The fullword binary item has a fullword alignment requirement; the character has a byte alignment requirement. In PL/I, the structure would be declared:

```
DCL 1  A,
    2  B CHAR (1),
    2  C FIXED BINARY (31,0);
```

and would be held thus:

In COBOL, the structure would be declared:

```
01 A,
   02  B,  PICTURE X, DISPLAY.
   02  C,  PICTURE S9(9), COMPUTATIONAL.
```

and would be held thus:

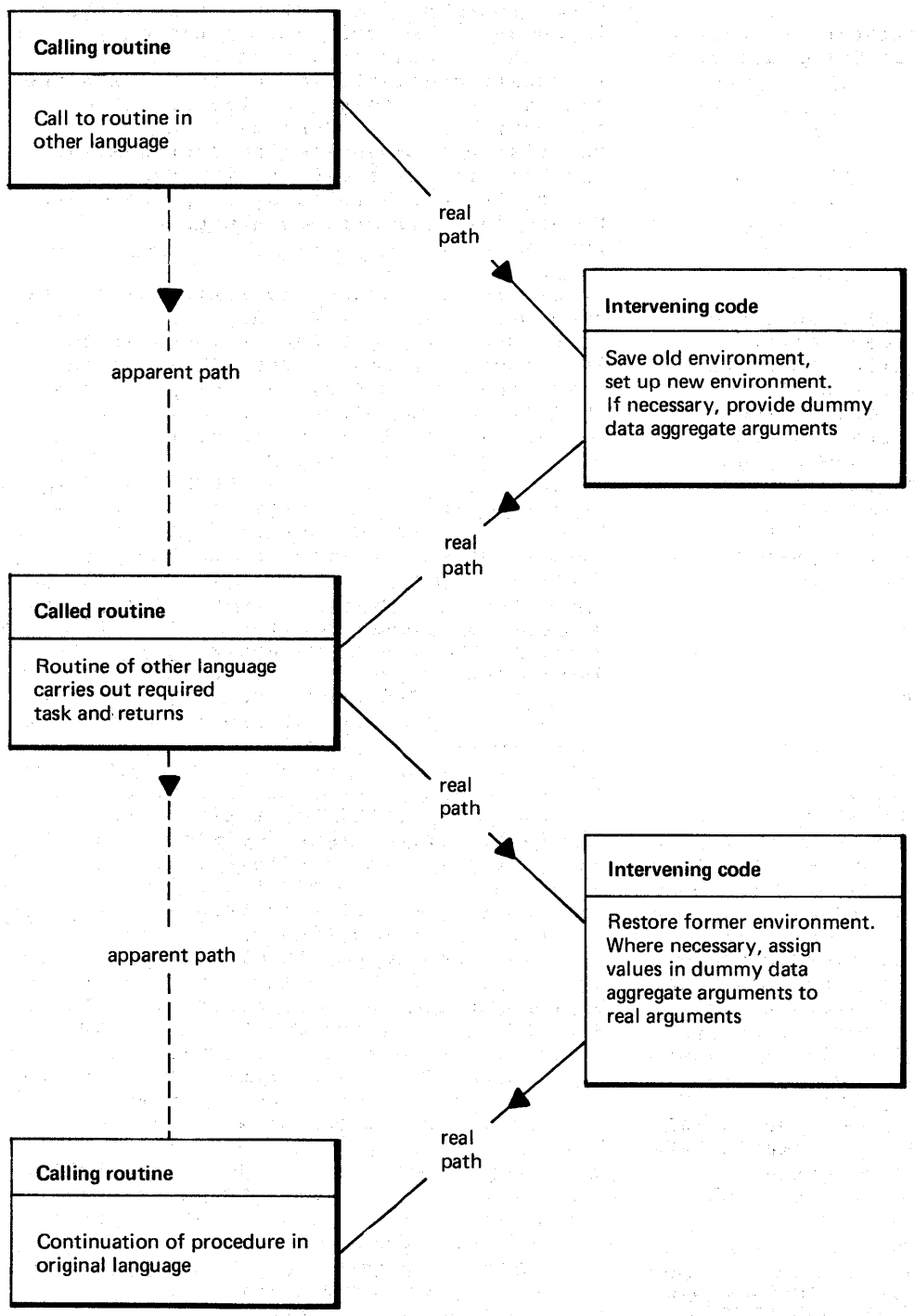


Figure 13.1. Principles of interlanguage communication

In FORTRAN, multidimensional arrays are held in column-major order. In PL/I, they are held in row-major order. Thus the second element in a FORTRAN two-dimensional array has the subscript (2,1), whereas the second element in a PL/I two-dimensional array has the subscript (1,2).

Structures are not available in FORTRAN. The equivalent of arrays in COBOL are held as in PL/I.

USE OF LOCATORS

When passing arguments, PL/I passes the address of locators for areas, arrays, strings, and structures rather than the address of the items themselves. This is because the routine that receives the arguments may require information about bounds or sizes of the data passed, and this is accessible through the locator. Other languages, however, expect the address of the data. The correct type of argument list must therefore be set up when an interlanguage call is made.

DIFFERENCES OF ENVIRONMENT

IBM FORTRAN compilers and the PL/I optimizing compiler rely upon initialization routines to set up an environment in which the compiled code routines can operate. In FORTRAN, the main task of the initialization routine is to issue a STXIT macro instruction to initiate the FORTRAN error-handling scheme. In PL/I, the initialization routines prepare for the PL/I error-handling scheme and also prepare the way for dynamic storage allocation. Register 12 is pointed at the TCA, which is used for addressing a number of housekeeping fields and library routines. Register 13 is pointed at a DSA which contains a standard save area, a NAB pointer pointing to the next available byte in the LIFO stack, and various other housekeeping fields. (See chapter 1 and chapter 5 for a discussion of the PL/I environment.)

When PL/I is called from either a COBOL routine or a FORTRAN routine, the PL/I environment must be set up before the program can be run. Similarly, when PL/I calls another language, the environment suitable for the program that has been called must be set up.

THE BASIC PRINCIPLES OF INTERLANGUAGE COMMUNICATION

The method used to solve the problems outlined above is to insert code immediately before and immediately after the execution of a routine in a different language. This code alters the environment and, where necessary, sets up dummy aggregate arguments to and from which the values can be assigned. The handling of the environment is done by three interlanguage housekeeping routines that are held in the resident library. Data aggregates are handled by compiled code. Figure 13.1 illustrates the basic principles.

The interlanguage facilities allow any number of calls to be made, and calls to both COBOL and FORTRAN routines can be made in the same program. Thus PL/I can call COBOL that calls PL/I that calls FORTRAN; FORTRAN can call PL/I that calls COBOL, and so on. All calls must, however, be made either to PL/I or from PL/I. Calls cannot be made directly between COBOL and FORTRAN. Options allow the programmer to specify that PL/I interrupt-handling facilities will be available through the COBOL or FORTRAN routines for those program checks that are not handled by COBOL or FORTRAN, and also allow the programmer to specify whether he wishes data aggregates to be automatically re-formatted when passed as arguments. (The programmer may wish to carry out the re-formatting himself.) The rules involved are fully described in the language reference manual. Briefly, they are as follows. For a PL/I procedure to call a COBOL or FORTRAN routine, the name of the routine must be declared as an external entry point with the option COBOL or FORTRAN in the OPTIONS attribute. If the programmer wishes to take advantage of the PL/I error-handling or interrupt-handling facilities, the INTER option must be included in the declaration. When a PL/I procedure is to be called by COBOL or FORTRAN, the keyword COBOL or FORTRAN should be included in the OPTIONS option of the PROC or ENTRY statement. To override the creation or remapping of dummy arguments for aggregates when calling FORTRAN or COBOL, or to override the creation or remapping of dummy parameters when being called from FORTRAN or COBOL, the NOMAP, NOMAPIN, and NOMAPOUT options can be used.

The compiler also allows the specification of the COBOL option in the ENVIRONMENT attribute of a PL/I file. This is separate from the interlanguage facilities described above, and is a method of allowing data sets produced by programs of one language to be used by programs of

SOURCE LISTING

```
1          P13P2:PROC;
2   1              DCL FRED OPTIONS(COBOL),
                       1 STRUCTURE,
                          2  C  CHAR (1),
                          2  D FIXED BINARY (31,0);
3   1            CALL FRED(STRUCTURE);
4   1          END;
```

OBJECT LISTING

```
* STATEMENT NUMBER  3
00006A  41 00 0 0C8              LA    0,8(0,0)
00006E  58 10 D 04C              L     1,76(0,13)
000072  1E 01                    ALR   0,1
000074  55 00 C 00C              CL    0,12(0,12)
000078  47 D0 2 018              BNH   CL.4
00007C  58 F0 C 048              L     15,72(0,12)
000080  05 EF                    BALR  14,15
000082                     CL.4   EQU   *
000082  50 00 D 04C              ST    0,76(0,13)
000086  41 11 0 000              LA    1,0(1,0)
00008A  50 10 D 0A8              ST    1,168(0,13)
00008E  D2 C3 D 088 D 0B3        MVC   WKSP.1+16(4),STRUC
                                       TURE.C
000094  58 80 D 088              L     8,WKSP.1+16
000098  D2 C3 1 00C D 088        MVC   STRUCTURE-179(4),W
                                       KSP.1+16
00009E  58 70 D 0B4              L     7,STRUCTURE.D
0000A2  50 70 1 004              ST    7,4(0,1)
0000A6  58 F0 3 00C              L     15,A..IBMBIECA
0000AA  18 91                    LR    9,1
0000AC  05 EF                    BALR  14,15
0000AE  50 90 3 030              ST    9,48(0,3)
0000B2  96 80 3 030              OI    48(3),X'80'
0000B6  1B 55                    SR    5,5
0000B8  41 10 3 030              LA    1,48(0,3)
0000BC  58 F0 3 034              L     15,52(0,3)
0000C0  18 69                    LR    6,9
0000C2  05 EF                    BALR  14,15
0000C4  58 F0 3 010              L     15,A..IBMBIECC
0000C8  05 EF                    BALR  14,15
0000CA  D2 03 D 088 9 000        MVC   WKSP.1+16(4),STRUC
                                       TURE-179
0000D0  58 F0 D 088              L     15,WKSP.1+16
0000D4  D2 C3 D 0B3 D 088        MVC   STRUCTURE.C(4),WKS
                                       P.1+16
0000DA  58 60 9 004              L     6,4(0,9)
0000DE  50 60 D 0B4              ST    6,STRUCTURE.D
```

Get VDA for dummy arguments

Place new value in NAB

Move structure into dummy

Branch to interlanguage housekeeping routine

Set up argument list

Branch to COBOL routine

Branch to interlanguage housekeeping routine

Move values from dummy to real arguments

Figure 13.2. Typical code when PL/I calls COBOL or FORTRAN routine

the other language. The use of the COBOL option in the ENVIRONMENT attribute is described in the last section of this chapter.

PL/I Calls FORTRAN or COBOL

When the calling program is PL/I, the compiler generates in-line code and library calls to handle the environment and data aggregate problems and places the code before and after the call to a program of a different language. The order of events is:

1. Re-arrange data aggregate arguments, if necessary, by creating and initializing a dummy of the correct format.
2. Call the appropriate interlanguage housekeeping routine to save the PL/I environment and prepare for the other program.
4. Call the required COBOL or FORTRAN routine passing a parameter list which does not use locators.
4. Call the interlanguage housekeeping routine to restore the PL/I environment.
5. If necessary, assign the values of the dummy data aggregate to the PL/I data aggregate.
6. Continue processing in the normal manner.

A typical code sequence illustrating the above process is shown in figure 13.2.

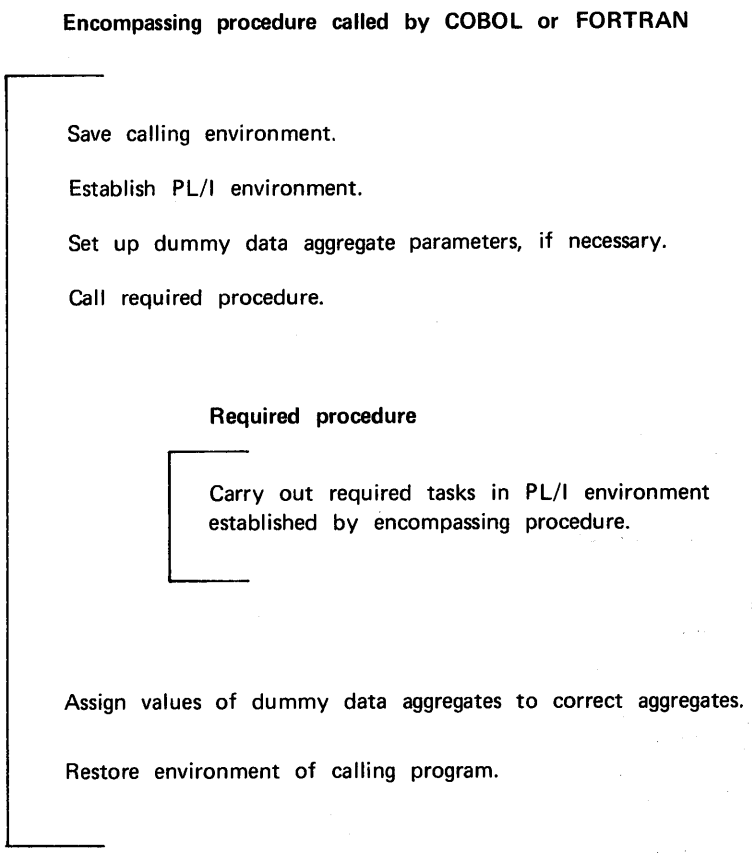


Figure 13.3. Nested procedures used when COBOL or FORTRAN calls PL/I

FORTRAN or COBOL Calls PL/I

When PL/I is called from another language, the method used is different. The code to handle environment and data aggregate problems cannot be included in the calling program, as this has been compiled in the normal manner by a COBOL or FORTRAN compiler. Instead the code is placed in the called program. This is done by the compiler generating two nested PL/I procedures. The outer procedure is the one that is actually called by the other program. It carries out the housekeeping duties, calling the interlanguage housekeeping routines to set up or restore the PL/I environment, and producing suitable dummy aggregate parameters if necessary. The inner procedure is compiled in the normal manner, called by the outer procedure, and executes the object code corresponding to the PL/I program. Throughout this chapter the outer procedure is called the encompassing procedure and the inner procedure the required procedure. The system of nested procedures is illustrated in figure 13.3.

Throughout the remainder of this discussion, the first procedure entered in a job or jobstep, which would be known in PL/I as the main procedure, is referred to as the principal procedure. This is because there is no COBOL or FORTRAN term equivalent to "main procedure" in PL/I.

When PL/I is called from COBOL or FORTRAN, the PL/I environment may already have been set up and may need restoring from the information that has been saved. This can happen in two circumstances.

1. The principal procedure in the job or jobstep may have been a PL/I main procedure which called FORTRAN or COBOL, which in turn called PL/I.

2. There may have been a previous call to PL/I in a job or jobstep whose principal procedure is in COBOL or FORTRAN. In this situation, the PL/I environment is retained until the calling routine itself is completed. This speeds execution of other calls to PL/I routines.

When the PL/I environment has already been established, it can be restored by pointing registers 12 and 13 at the TCA and

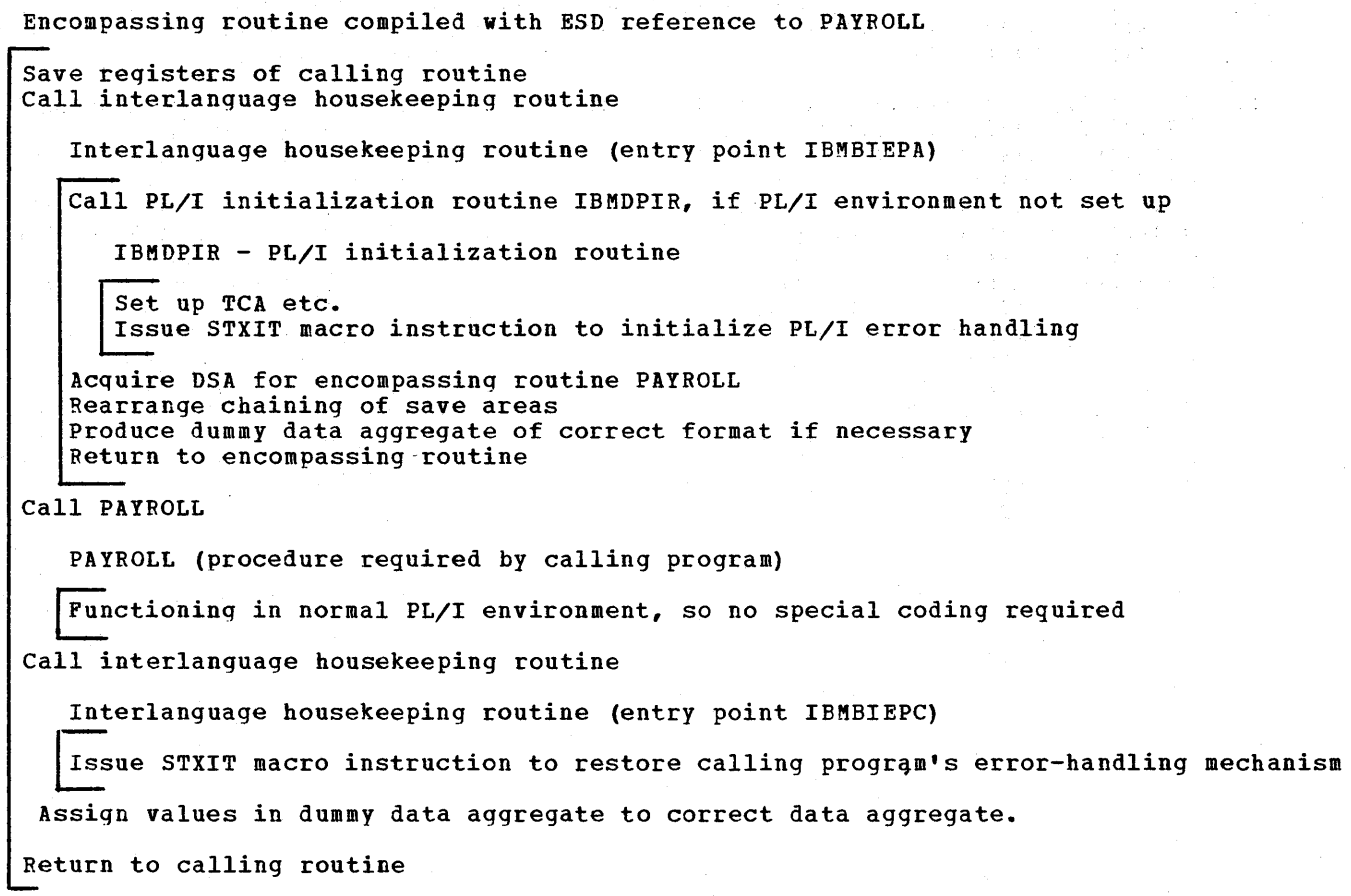


Figure 13.4. Action when setting up PL/I environment on call from COBOL or FORTRAN principal procedure

the current DSA respectively, and resetting the program check exit and program mask so that program checks are passed to the PL/I condition- handling modules. However, on the first call in a program with a principal procedure in COBOL or FORTRAN, the PL/I environment must be completely initialized. This involves acquiring storage for the program management area and for dynamic storage allocation. This storage is known as the ISA(initial storage area) and is described in chapter 6.

The area used for the ISA will be that part of the partition that is not taken up by the executable program phase, unless an area has been assigned, in either a COBOL or FORTRAN routine, by use of a call to PLISA. (PLISA is described later in this chapter.) Space is also allowed in the high-address end of the partition for the DTF and buffers for SYSLST.

Sequence of Events when PL/I is called from FORTRAN or COBOL: When PL/I is called from FORTRAN or COBOL, the routine that gets control is the encompassing PL/I routine. This routine is given the name of the procedure called from COBOL or FORTRAN, and appropriate ESD references are generated. The subsequent sequence of events depends on whether or not the PL/I environment has been previously initialized. The sequence is given below and illustrated in figures 13.4 and 13.5.

1. The encompassing routine is called by COBOL or FORTRAN and:
   a. Saves the registers of the calling program.
   b. Calls the interlanguage housekeeping routine, passing the interlanguage routine the size of the DSA that the encompassing routine itself will require.
2. The interlanguage housekeeping routine then:
   a. Tests to see if the PL/I environ- has been established previously and can be restored.

b. If possible, restores the PL/I environment and returns to the encompassing routine.

c. If the PL/I environment has yet to be initialized, the housekeeping routine calls IBMDPIR, passing it an address in the interlanguage housekeeping routine, to which control will return. Control blocks are set up to handle the housekeeping problems. Save areas are rechained so that the save area of IBMDPIR (dummy DSA) comes before the save area for the COBOL or FORTRAN calling program. Consequently, the PL/I environment will not be lost until the calling program itself is finished. Module IBMDIEP inserts a short save area and an interlanguage save area, and IBMDPII creates a dummy DSA, all of which are left in the DSA chain (see figure 13.5). These save areas are specially created and used to return control to the interlanguage housekeeping routine before and after the execution of IBMDPIR on termination. Control is then returned to the encompassing procedure.

3. The encompassing procedure reformats data aggregate arguments if necessary, sets up locators where they will be expected by PL/I, and calls the required PL/I routine.

4. The required PL/I routine carries out the required operations and returns control to the encompassing procedure.

5. The encompassing procedure reassigns the data aggregate arguments, if any, and calls the interlanguage housekeeping routine.

6. The interlanguage housekeeping module:

   a. Saves the PL/I environment.

   b. Restores the environment of the calling program.

7. Control is returned to the encompassing routine, which, in turn, returns control to the original FORTRAN or COBOL calling procedure.

CONTROL BLOCKS IN INTERLANGUAGE COMMUNICATION

Three control blocks are used during interlanguage communication. They are used to indicate which environments have been established, and to save environment and interrupt information.

1. IBMBILC1 A control section included in each interlanguage housekeeping routine. The control section consists of two words. The first word contains a pointer to ZCTL(see below). The second word contains three flags: COBOL and FORTRAN flags indicate whether the COBOL or FORTRAN environment has been set up and still exists; the third flag is a stack flag which indicates whether a call has been made to PLISA to indicate where the ISA should be placed.

2. ZCTL A control block generated on the first interlanguage call and retained until the PL/I environment is discarded, or until the end of job. ZCTL is set up in the high-address end of the area used for the ISA. It is set up as non-LIFO dynamic storage when PL/I calls FORTRAN or COBOL. When PL/I is called from COBOL or FORTRAN, ZCTL is set up before any of the PL/I environment is established; however, it is in the position that would be occupied by non-LIFO dynamic storage, although it is not in the ISA.

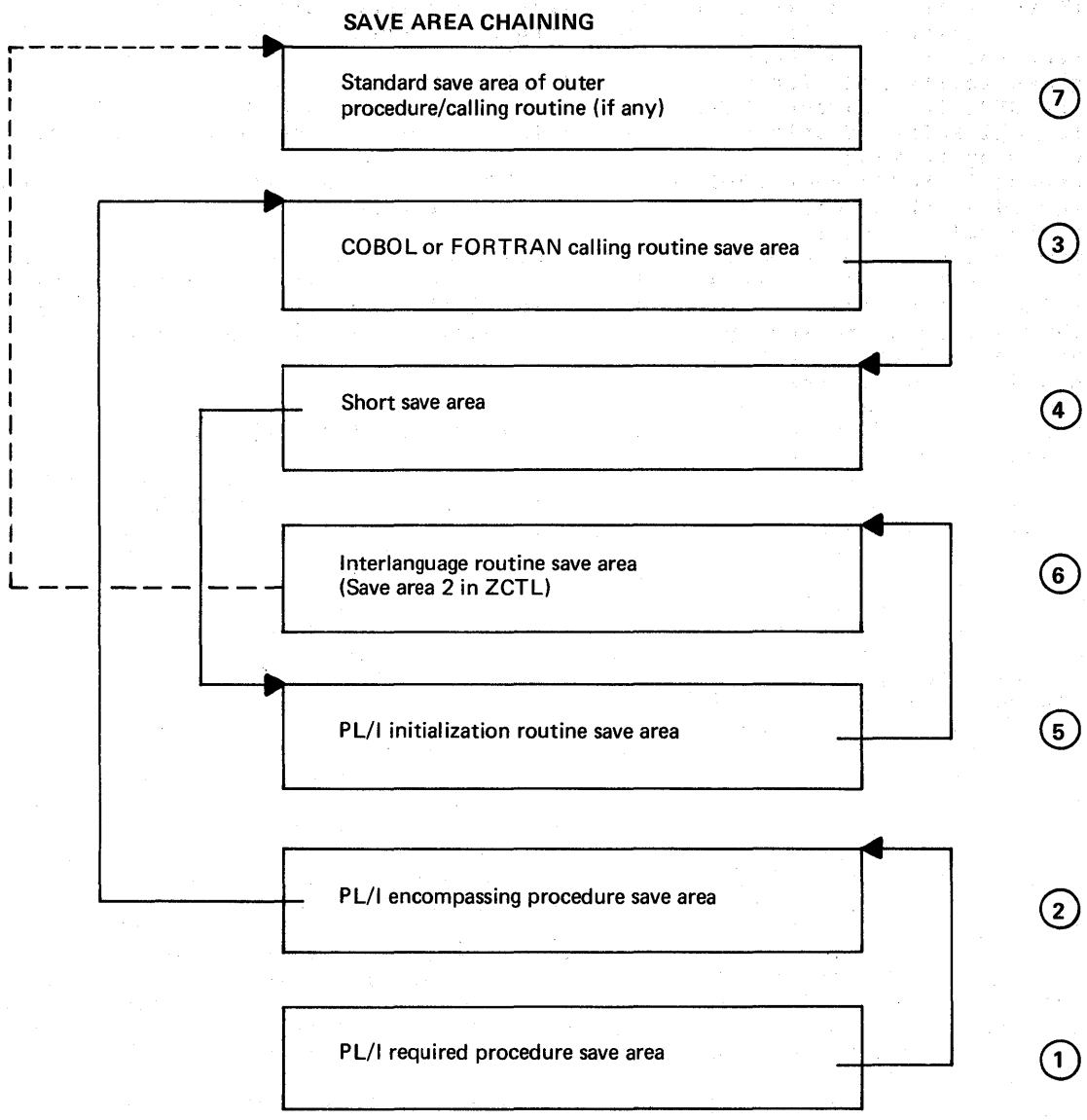
3. Interlanguage VDA

   This is a control block that is generated in a VDA in the LIFO stack for every call to COBOL or FORTRAN, or initializing call to PL/I. It is used to retain register 13 and to retain COBOL and FORTRAN interrupt information during the execution of nested calls.

Figures 13.6 and 13.7 show how these control blocks are used in a series of interlanguage calls that start from PL/I and FORTRAN principal procedures respectively.

SPACE FOR PL/I DYNAMIC STORAGE AND PROGRAM MANAGEMENT AREA

Unlike FORTRAN or COBOL, PL/I requires space for dynamic storage allocation and



Rearrangement of save area chaining takes place after the first call to PL/I, so that the PL/I environment is not discarded until the calling routine itself is finished.

Save areas that return control to the PL/I initialization routine and interlanguage housekeeping routine are placed before the calling routine. (The numbers 1-7 in the diagram show the order of backchaining).

Figure 13.5. Chaining of save areas when PL/I is called from COBOL or FORTRAN principal procedures.

for a program management area. These areas are an important part of the PL/I environment and are set up during initialization of the ISA.

The default action of the PL/I initialization routine is to set up the ISA in that part of the partition that is not taken up by the executable program phase, allowance being automatically made for a buffer and a DTF for SYSLST. Consequently, there is no need to call PLISA if only SYSLST is used for I/O by FORTRAN. However, the programmer in FORTRAN or COBOL has a method of overriding this action by a call to PLISA. In the call to PLISA, a FORTRAN or COBOL variable must be named and a length given. The ISA will then be set up at an address starting at the first doubleword in the variable, and taking up an area large enough to hold the specified length starting on a doubleword. This length should not be greater than that of the variable, or other parts of the COBOL or FORTRAN program will be overwritten by the ISA.

This facility is necessary because FORTRAN I/O buffers use space outside the executable program phase and can consequently overwrite or be overwritten by the ISA. However, when the area to be used for the ISA is included in the executable program phase, as it will be if a FORTRAN or COBOL variable is used, the area will be automatically protected from overwriting. The facility is available in COBOL programs, since these programs may call PL/I, which may in turn call FORTRAN. Since the ISA is set up on the first call to PL/I, the problem of overwriting with FORTRAN buffers arises in this situation, unless the area is specified before the first call to PL/I.

A call to PLISA goes to a section of the interlanguage housekeeping routine of which PLISA is an alias. This routine sets a flag in IBMBILC1, to indicate that an area has been designated for the ISA, and alters a parameter list for IBMDPIR in such a way that the length and address of the largest area that can be bounded with doubleword boundaries inside the declared length are placed in the parameter list.

When the PL/I interlanguage housekeeping routine IBMDIEP is called, it tests to see if the stack flag is on. If the flag is on, the ISA is set up in the area designated in the PLISA call.

Handling Changes of Environment

Because the environments required for the various languages differ, they are handled by three distinct library modules. These modules are known as interlanguage housekeeping modules.

Three modules are involved in the management of housekeeping during interlanguage communication.

1. IBMDIEC: COBOL when called from PL/I
2. IBMDIEF: FORTRAN when called from PL/I
3. IBMDIEP: PL/I when called from FORTRAN or COBOL.

Each module has a number of entry points to deal with various situations, and each is called immediately before and immediately after the program that is required.

COBOL WHEN CALLED FROM PL/I (IBMDIEC)

When calling COBOL, IBMDIEC carries out the following tasks:

Before Entry to COBOL Program (IBMBIECA, IBMBIECB)

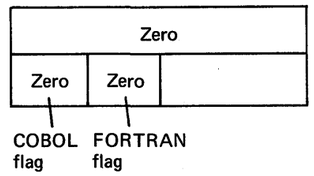
1. Test to see if this is the first interlanguage call; if so, set COBOL flag in IBMBILC1 and set up ZCTL.
2. Acquire interlanguage VDA and store register 12 in ZCTL, register 13 in the VDA. Write null (zero) error information in ZCTL.
3. If INTER option not specified (i.e., entry point IBMBIECA), issue STXIT macro instruction and set program mask so that errors will be handled by the supervisor. Return to compiled code.
4. If INTER option is specified (entry point IBMBIECB), issue new STXIT macro instruction and return.

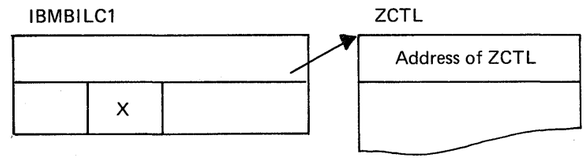
On Return from COBOL Program (IBMBIECC)

The following actions take place on return:

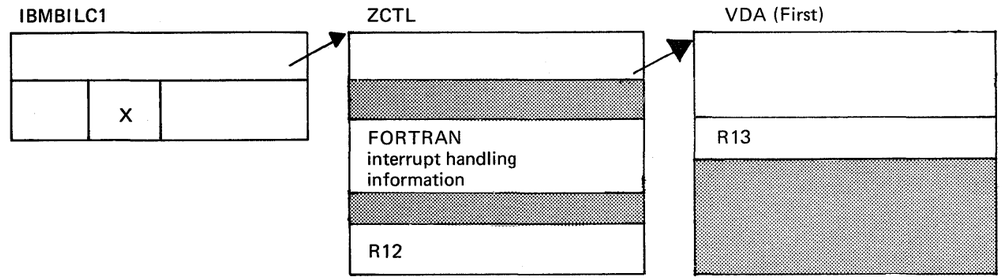
1. A STXIT macro instruction is executed, which results in the program check exit being set to pass control directly to the PL/I interrupt handler.
2. The first word of the interlanguage

1 **Initial situation**
IBMBILC1 is set up as a control section by the PL/I interlanguage routines. Its first word and flags are initially zero.



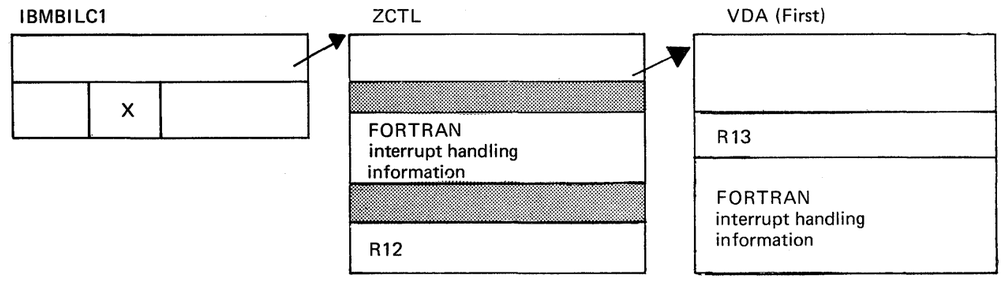


2 **Call FORTRAN from PL/I (IBMBIEF)**
The compiler generates a call to the interlanguage communications routine. This routine:
1. Sets up ZCTL after testing for zero pointer in IBMBILC1. Acquires an interlanguage VDA.
2. Sets ZCTL pointer to interlanguage VDA, and IBMBILC1 pointer to ZCTL.
3. Sets FORTRAN flag in IBMBILC1. Saves R12 in ZCTL, R13 in interlanguage VDA.
4. Calls FORTRAN library to initialize FORTRAN SPIE
5. Resets program check exit as required.
6. Returns to compiled code, which calls FORTRAN procedure.



3 **Call PL/I from FORTRAN (IBMBIEP)**
The PL/I program, because it is declared with the option FORTRAN, will have been compiled inside an encompassing procedure. The encompassing procedure is the one called by FORTRAN. The encompassing procedure calls the interlanguage communications routine IBMBIEP, which:
1. Checks IBMBILC1 to see if either FORTRAN or COBOL flag is set. As one flag is set, restores registers.
2. Issues PL/I SPIE and STAE and stores interrupt handling information of calling program in interlanguage VDA.

Control then returns to the encompassing program, which calls the required PL/I program.



4 **Call COBOL from PL/I (IBMBIEC)**
The PL/I program will contain a call to the interlanguage routine IBMBIEC, which:
1. Sets up another interlanguage VDA, points ZCTL to this VDA, and places the old value of ZCTL's pointer in the VDA.
2. Stores R13 in the new VDA.
3. Issues a SPIE so that error handling will be as requested by PL/I program.

Control is then returned to compiled code, which then calls the COBOL routine.

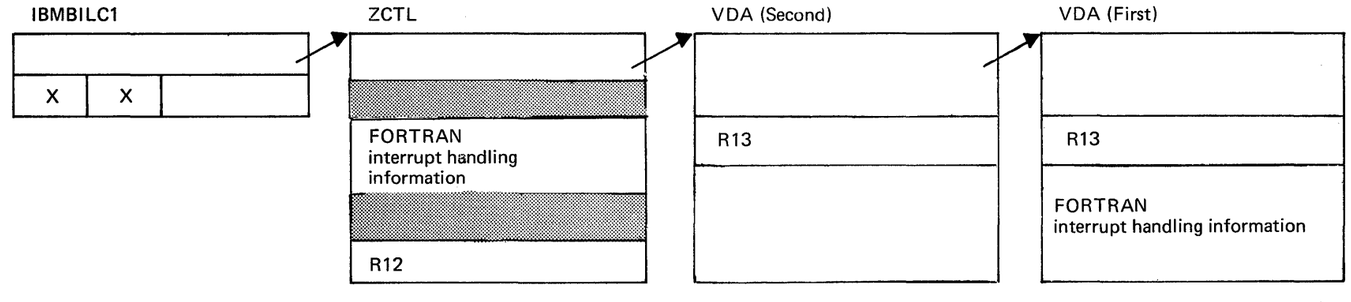
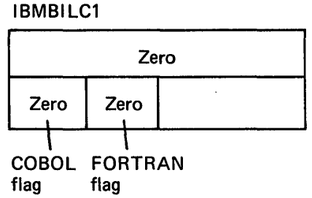
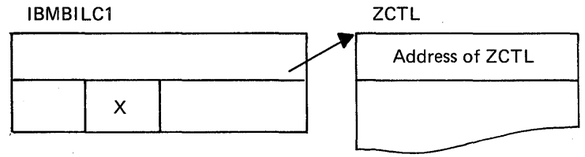


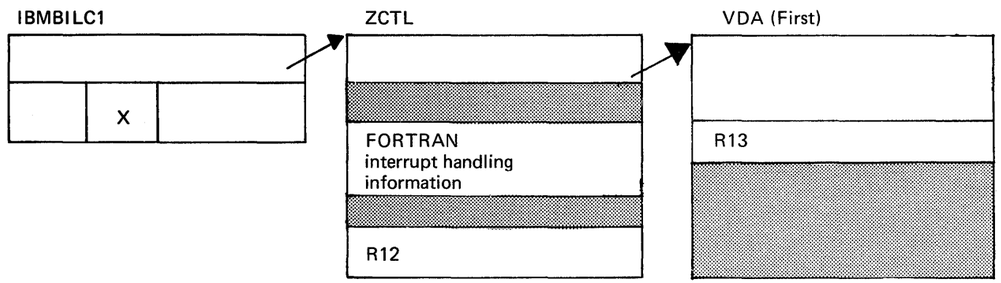
Figure 13.6. Example of chaining sequences (PL/I principal procedure)





Final situation **8**

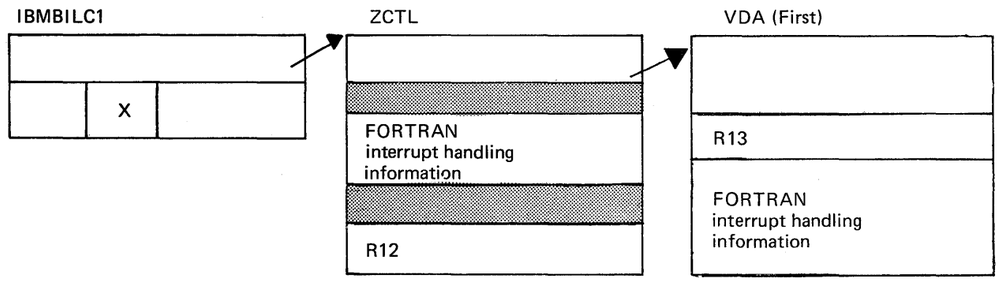
ZCTL is retained until program is completed.



FORTRAN returns to PL/I (IBMBIEF) **7**

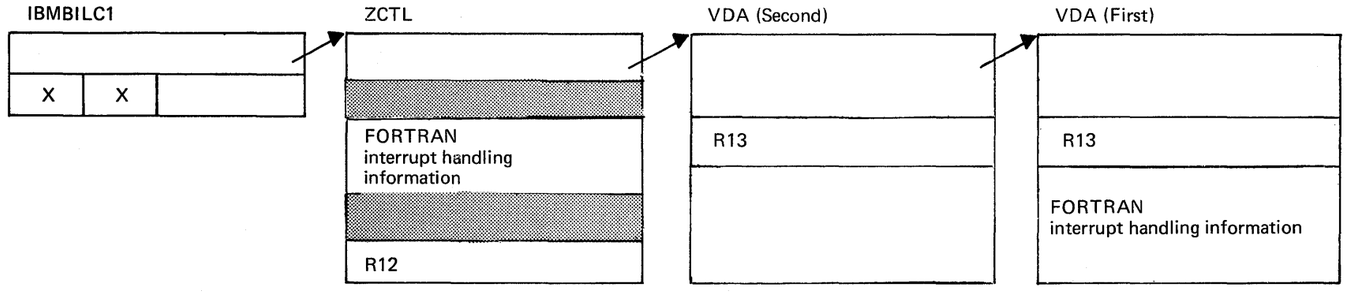
When control returns from a FORTRAN procedure, a call is made to the interlanguage communication routine IBMBIEF, which:

1. Moves the pointer in the VDA to the first word of ZCTL.
2. Issues a PL/I SPIE macro.
3. Issues a PL/I STAE macro leaving the previously-stacked FORTRAN STAE for possible future use.
4. Returns control to compiled code.



PL/I returns to FORTRAN (IBMBIEP) **6**

When the required PL/I procedure is finished, it returns control to the encompassing procedure. The encompassing procedure calls the interlanguage routine IBMBIEP, which issues a SPIE macro instruction to restore the error-handling situation to that of the calling routine. The information for the SPIE macro instruction is retrieved from the interlanguage VDA. The current PL/I STAE is canceled, leaving the previously stacked FORTRAN STAE in control.
The interlanguage routine returns control to the PL/I encompassing procedure, which then returns control to the FORTRAN program.



COBOL returns to PL/I **5**

The COBOL program returns to the PL/I program, which immediately calls the interlanguage routine IBMBIEC. This routine rearranges the chain by placing the word in the most recent VDA in the first word of ZCTL. It then issues a SPIE macro instruction to restore the PL/I error-handling situation.

PL/I compiled code then continues.

al procedure)

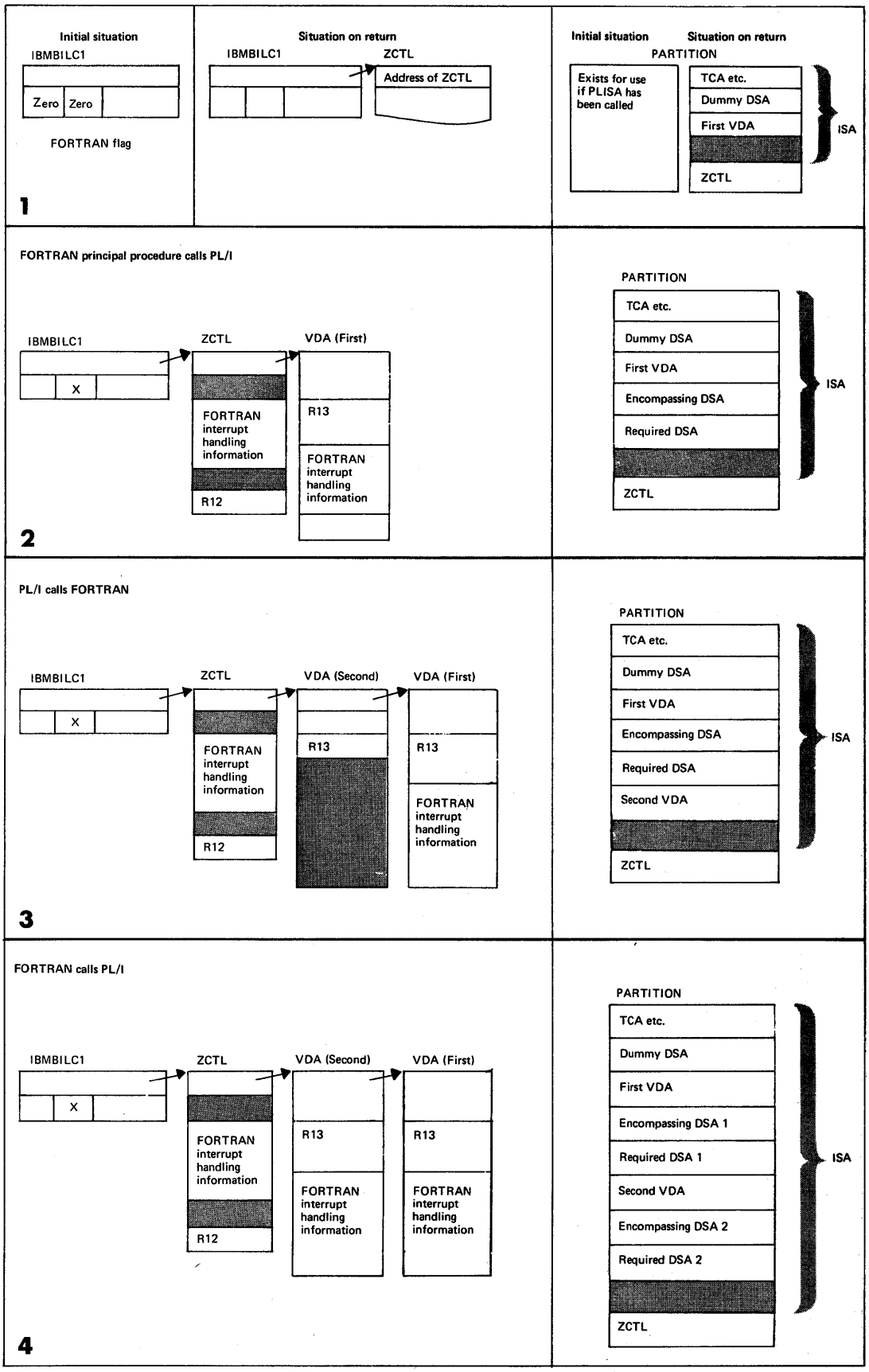


Figure 13.7. Example of chaining sequences (FORTRAN principal procedure)

VDA is moved into the first word of ZCTL, and the VDA is freed.

Action on Interrupt in COBOL

If the INTER option is not specified, all program checks will be handled by the supervisor or the COBOL library. However, if the INTER option is specified and the COBOL program has been compiled with a request for the COBOL interrupt handler not to be called, the following takes place.

1. During the first invocation of IBMDIEC, a STXIT macro instruction is issued, which results in interrupts being passed to an entry address in IBMDIEC.

2. When an interrupt occurs, register 12 is restored from ZCTL and register 13 from the interlanguage VDA, thus restoring the PL/I environment.

3. A DSA is acquired for IBMDIEC in LWS. The address of the interrupt, in the second word of the PSW, is saved in this DSA and replaced by the address of another entry address in IBMDIEC. For underflow interrupts, the four bytes preceding the point of interrupt are also copied and placed before the entry address in case the error handler needs to examine them. This point acts as the return address for the PL/I error handler.

4. Flags are set in the TCA and DSA to indicate that it is possible for an abnormal GOTO to occur in a PL/I on-unit.

5. A STXIT macro instruction is issued to transfer the program check exit to the PL/I error-handling routines whose address is held in the TCA appendage.

6. An interrupt is then caused, and control is passed to the PL/I error-handling routines by the supervisor.

Return from Interrupt

If a normal return to the point of interrupt is made, the following takes place.

1. When the PL/I error-handling routines return control to what they take to be the point of interrupt, control in fact returns to an entry address in IBMDIEC.

2. A further STXIT macro instruction is issued altering the program check exit to a further point in IBMDIEC. An interrupt is then caused and control passed through the supervisor to the new interrupt address. The reason for this is that the program in which the error occurred expects all registers to be restored, and this can only be done if return is made by the LPSW instruction. This is eventually caused by the EXIT macro. The address of the interrupt, taken originally from the second word of the PSW, is then restored to the PSW which has been saved in the area nominated by the STXIT macro instruction. The COBOL registers are restored to the interrupt save area. The program check exit is altered by a further STXIT macro instruction to IE007.

3. Control is returned to the supervisor by an EXIT macro instruction, which returns control to the point of interrupt.

If, however, return occurs via the abnormal GOTO mechanism, IE015 branches to IE018, which unchains and frees the latest VDA and returns to the abnormal GOTO code.

FORTRAN WHEN CALLED FROM PL/I (IBMDIEF)

When FORTRAN is called by PL/I, the module IBMDIEF is entered immediately before and immediately after the execution of the FORTRAN program. The processing done before entry to the FORTRAN program depends on whether the INTER option is specified. Entry point IBMBIEFA handles calls without the INTER option. Entry point IBMBIEFB handles calls with the INTER option.

Before Entry to FORTRAN Program (IBMBIEFA and IBMBIEFB)

Prior to the call to FORTRAN, IBMDIEF does the following:

1. Tests the pointer in IBMBILC1 to discover if this is the first interlanguage call. If it is the first call, it sets up ZCTL and sets the FORTRAN flag in IBMBILC1. If it is not the first call, it tests to see whether the FORTRAN flag is set in IBMBILC1 and sets the FORTRAN flag if it is not already set.

2. If the FORTRAN environment has not previously been set up, calls the

FORTRAN initialization routine. This routine sets up the program check exit so that program interrupts will be handled by the FORTRAN error handling method. The FORTRAN error data is stored in ZCTL.

3. Acquires an interlanguage VDA. Points the first word of ZCTL to this VDA, taking the value previously in the first word of ZCTL and placing it in the first word of the VDA. (This places the new VDA at the head of a chain starting from ZCTL.)

4. Stores PL/I's register 13 in the interlanguage VDA., thus saving the PL/I environment.

5. If INTER option is not specified issues a FORTRAN STXIT macro instruction from ZCTL, sets program mask to '2', and returns to compiled code.

6. If INTER option is specified, a STXIT macro instruction is issued that will result in control being passed to an entry address in IBMDIEF, should an interrupt occur. The program mask is reset to 'E' in case it was changed by the FORTRAN initialization routine.

Action on Return from FORTRAN Program (IBMBIEFC and IBMBIEFD)

When return is made from the FORTRAN subroutine, PL/I compiled code immediately makes a call to the FORTRAN interlanguage routine. If the FORTRAN routine may be used as a function, entry point IBMBIEFD is used. Otherwise, entry point IBMBIEFC is used. The module IBMDIEF does the following:

1. A STXIT macro instruction is issued that resets the program check exit to the PL/I error-handling modules, and the program mask is set to 'E'.

2. The first word of the interlanguage VDA is placed in the first word of ZCTL and the VDA freed.

3. For entry point IBMBIEFD (the FORTRAN function entry point) the parameter list passed by PL/I is examined, and the values are moved out of the registers in which they were placed by the FORTRAN routine, and moved to the correct location.

Action on Interrupt in FORTRAN

If the INTER option is not specified, the action on any interrupt that occurs in the FORTRAN program will be that specified in the FORTRAN error-handling scheme. However, if the INTER option is specified, all program checks that are not handled by FORTRAN error-handling are passed to the PL/I error-handling modules.

The FORTRAN error-handling scheme is used after the following interrupts have occurred:

1. Specification (other than for invalid instruction address)

2. Fixed-point divide

3. Decimal divide

4. Exponent overflow

5. Exponent underflow

6. Floating-point divide

All other program checks are handled by the PL/I error handler.

When an interrupt occurs, the following takes place:

1. When control is passed by the supervisor to an entry address, the type of interrupt is discovered by examining the PSW. If the interrupt is one of the types that can be handled by FORTRAN, the normal FORTRAN environment is established and the FORTRAN error handling module invoked.

2. If it is not the type of interrupt that can be handled by FORTRAN, register 12 is restored from ZCTL and 13 from the latest interlanguage VDA.

3. The address of the interrupt is taken from the second word of the PSW and stored in the DSA. The second word of the PSW is then replaced by another entry address in IBMDIEF.

4. Flags are set in the TCA and DSA to indicate that it is possible for an abnormal GOTO to occur in a PL/I on-unit.

5. A STXIT macro instruction is then issued to restore the PL/I error-handling situation. A branch is then made to the PL/I error handler.

6. For a normal return, the PL/I or FORTRAN error-handling routine returns

to the point of interrupt, which it takes from the second word of the PSW. This, in fact, is the entry address in IBMDIEF, which has been placed in the PSW in the PL/I interrupt save area. (See 3 above)

7. If, however, return occurs via the abnormal GOTO mechanism, control passes to an address in IBMDIEF that unchains and frees the latest VDA and returns to the abnormal GOTO code.

8. A STXIT macro instruction is issued to alter the program check exit to a third address in IBMDIEF.

9. An interrupt is then caused, and the supervisor passes control to the program check exit address set in 8 above.

10. A further STXIT macro instruction is then given to restore the program check exit to the position at the start of the process.

    The method described in 7, 8, 9, and 10 above is adopted as control has to be returned via the supervisor so that the values of all registers may be restored.

11. The word originally taken from the PSW and stored in the DSA is restored to the PSW, which now holds the address of the point of interrupt. The FORTRAN registers are restored to the save area and an EXIT macro issued. This results in control returning through the supervisor to the point of interrupt.

PL/I CALLED FROM COBOL OR FORTRAN (IBMDIEP)

As with the other interlanguage communication routines, IBMDIEP is called immediately before and immediately after the program that is to be executed. However, the interlanguage housekeeping routine cannot be called direct from the COBOL or FORTRAN routine, because the existence of such a routine is unknown to COBOL or FORTRAN. To overcome this problem, an encompassing routine is generated with the same name as the PL/I routine. This encompassing routine is called by COBOL or FORTRAN and in turn calls the interlanguage housekeeping routine and the required PL/I routine. Code generated for a typical encompassing routine is shown in figure 13.8.

Although the names of both PL/I procedures are the same, the encompassing routine gets control when called from COBOL or FORTRAN, because no ESD records are generated for the interlanguage entry points of the required PL/I program.

Before Entry to PL/I program (IBMBIEPA)

Before a call is made to the PL/I program, IBMDIEP does the following:

1. Tests to see if the PL/I environment has already been initialized, by examining whether the COBOL or FORTRAN flag in IBMBILC1 is set.

2. If the COBOL or FORTRAN flag is on, this means that a previous interlanguage call has been made, and as the call must have been made either to or from PL/I, the PL/I environment must have been set up. Register 12 is restored from ZCTL. A STXIT macro instruction is issued so that program checks are handled by the PL/I condition handler.

   The FORTRAN flag is set on and control returned to the PL/I encompassing procedure.

3. If neither the COBOL nor the FORTRAN flag is on, PL/I is being called for initialization by a program whose principal procedure is in COBOL or FORTRAN.

   The following actions take place:

   a. IBMDIEP sets up ZCTL and then calls the initialization/termination routine IBMDPIR to set up the PL/I environment. It passes the address of the storage to be used as an ISA. This is either the storage specified in a call to PLISA or the section of the partition between the executable program phase and an area allowed for the DTF and the buffer for SYSLST. IBMDPIR is passed an address within IBMDIEP.

   b. IBMDPIR when completed makes a call to the entry point of IBMDIEP it was passed. This entry point saves the registers of IBMDPIR and rearranges the register save areas. The chaining of save areas is altered so that the save area that returns control to the initialization/termination routine IBMDPIR is placed above the save area of the routine that called the PL/I program.

As this rearrangement could cause certain housekeeping problems two additional save areas are created and inserted in the chain before and after the save area for the initialization termination routine. Restoring the registers of these save areas results in control passing to IBMDIEP which handles any housekeeping problems. These two save areas are known as "save area two" and "the short save area".

c. The FORTRAN or COBOL flag is set depending on the language of the calling program.

d. A DSA for the PL/I encompassing routine is acquired and its address returned to the encompassing routine.

The encompassing procedure then points register 13 to its DSA, and after any necessary re-formatting of parameters calls the required PL/I routine.

The order in which save areas are held starting with the oldest is:

Caller's caller

Save area two

Dummy DSA (save area for IBMDPIR)

Caller's save area (save area for COBOL or FORTRAN calling routine

Short save area

PL/I encompassing procedure DSA

PL/I required procedure DSA

4. A DSA for the encompassing routine is acquired.

5. Control is then returned tp compiled code in the encompassing routine.

Action after the PL/I Program is Completed (IBMBIEPC and IBMBIEPD)

IBMDIEP is called at the end of the PL/I routine by the encompassing routine generated by the compiler. If the calling program is FORTRAN, a returned value may be expected in register 0 or one or more of the floating-point registers. When this is the case, the entry point IBMBIEPD is used and the returned value is loaded into the required position. In other situations, the entry point IBMBIEPC is used. The module resets the program mask and issues a STXIT macro instruction to restore the calling routine's program check exit, the address of which has been stored in the interlanguage VDA.

Interrupt Handling

When PL/I is called by COBOL or FORTRAN, error handling is carried out in the normal PL/I manner. The STXIT macro instruction is issued by IBMDPII when the PL/I environment is first set up. For calls after the first, the STXIT macro instruction is issued by IBMDIEP.

Handling Data Aggregate Arguments

In order to communicate effectively between COBOL and PL/I, and FORTRAN and PL/I, a method of handling data aggregate arguments is necessary, because the three languages hold data aggregates in different ways.

ARRAYS

Arrays as such are not used in COBOL. The use of OCCURS in structures does, however, have a similar effect. However, PL/I structures of arrays and COBOL structures using OCCURS are both held in row-major order. In FORTRAN, arrays are held in column-major order. Thus, in a two-dimensional array, the element known in the FORTRAN array as (2,1) will become (1,2) in the PL/I array.

STRUCTURES

Structures are not used in FORTRAN. In COBOL the alignment requirements are met differently from PL/I. Full details of the differences in mapping are given in the language reference mannual for this compiler.

METHOD USED

The method used in handling data aggregates is to create dummy arguments of the correct format and let the called routine use the dummy. The values in the dummy are then

```
 OBJECT LISTING

000000                                 DC     C' P13P11'
000007                                 DC     AL1(6)

* INTERLANGUAGE PROCEDURE                     P13P11

* REAL ENTRY
000008  90 EC D 00C                   STM    14,12,12(13)          Store registers
00000C  47 F0 F 014                    B      *+16
000010  00000000                       DC     A(STMT. NO. TABLE)
000014  000000A0                       DC     F'160'
000018  00000000                       DC     A(STATIC CSECT)
00001C  58 30 F 010                    L      3,16(0,15)            Set R3 as static base
000020  41 10 0 004                    LA     1,4(0,0)
000024  58 00 F 00C                    L      0,12(0,15)            Pass length required for DSA
000028  18 8F                          LR     8,15                  Retain entry point address
00002A  58 F0 3 018                    L      15,A..IBMBIEPA        Branch and link to interlanguage housekeeping routine
00002E  05 EF                          BALR   14,15
000030  18 F8                          LR     15,8                  Restore entry point address to R15
000032  D2 03 D 054 3 030              MVC    84(4,13),48(3)        Set up on-unit flags
000038  58 10 D 004                    L      1,4(0,13)
00003C  58 10 1 018                    L      1,24(0,1)
000040  D2 0B D 078 1 000              MVC    120(12,13),0(1)       Place parameters at head of temporary storage
000046  92 00 D 080                    MVI    128(13),X'00'
00004A  05 20                          BALR   2,0                   Set R2 as program base

* PROCEDURE BASE
00004C  58 90 D 078                    L      9,120(0,13)           Point R9 at arguments
000050  50 90 3 038                    ST     9,56(0,3)             Store in argument list
000054  58 80 D 07C                    L      8,124(0,13)           Place address of fullword in argument list
000058  50 80 3 03C                    ST     8,60(0,3)             for possible returns value
00005C  58 70 D 080                    L      7,128(0,13)
000060  50 70 3 040                    ST     7,64(0,3)
000064  41 60 D 098                    LA     6,152(0,13)
000068  50 60 3 044                    ST     6,68(0,3)
00006C  96 80 3 044                    OI     68(3),X'80'           Mark end of argument list
000070  18 55                          SR     5,5                   Set static backchain to zero
000072  41 10 3 038                    LA     1,56(0,3)             Point R1 at parameter list
000076  58 F0 3 008                    L      15,A..P13P11          Branch and link to required procedure
00007A  05 EF                          BALR   14,15
00007C  41 60 D 098                    LA     6,152(0,13)           Pick up RETURNS value
000080  50 60 3 048                    ST     6,72(0,3)             Store in static storage
000084  41 10 3 048                    LA     1,72(0,3)             Point R1 at returns value
000088  58 F0 3 01C                    L      15,A..IBMBIEPD        Branch and link to interlanguage housekeeping routine
00008C  05 EF                          BALR   14,15
00008E  58 D0 D 004                    L      13,4(0,13)
000092  58 E0 D 00C                    L      14,12(0,13)           Restore all registers except R1 (used for returns value)
000096  98 2C D 01C                    LM     2,12,28(13)
00009A  07 FE                          BR     14                    Return to caller

* END INTERLANGUAGE PROCEDURE
```

Figure 13.8. Encompassing procedure to be called by FORTRAN

assigned to the original argument when the execution of the called program is completed.

If the data aggregates are nonadjustable, the mapping will be done during compilation and both the PL/I and the COBOL or FORTRAN mapping are produced. If the data aggregates are adjustable, the mapping is done during execution. Before the execution of the call to a program in another language, the data is transferred into the correctly mapped aggregate, which will be held in PL/I temporary storage. The values are reassigned to the original data aggregate after execution of the program in the other language.

The assignment of data between the dummy and the argument is done by compiled code.

NOMAP, NOMAPIN, AND NOMAPOUT OPTIONS

The NOMAP, NOMAPIN, and NOMAPOUT options can be used by the programmer to specify that data aggregates will not be remapped and placed in dummy arguments.

When NOMAP is specified, or when both NOMAPIN and NOMAPOUT are specified, the dummy is not generated at all, and the structure or array is passed as it stands.

When only NOMAPIN is specified, a dummy is created, but it is not initialized with the values of the aggregate being passed. However, on return from the COBOL or FORTRAN routine, the data in the dummy is placed in the data aggregate that is being passed.

When only NOMAPOUT is specified, a dummy is created, and the data from the data aggregate is moved into the dummy. When control is returned to the calling program, however, the data from the dummy is not moved into the data aggregate that was passed.

CALLING SEQUENCE

When PL/I calls COBOL or FORTRAN passing data aggregates as arguments, the sequence of events is:

1. Handle data reassignment to dummy by compiled code.
2. Call interlanguage housekeeping routine.
3. Call COBOL or FORTRAN routine.
4. Call interlanguage housekeeping routine.
5. Assign data in dummy to real argument, by means of compiled code.

When COBOL or FORTRAN calls PL/I, the sequence of events is:

1. The COBOL or FORTRAN routine calls the encompassing PL/I routine.
2. The encompassing PL/I routine:
   a. Calls the interlanguage housekeeping routine.
   b. Sets up the necessary dummy data aggregate argument by compiled code.
   c. Calls the required PL/I routine.
   d. Reassigns the data from the dummy by compiled code.
   e. Calls the interlanguage housekeeping routine.
   f. Returns to the original calling routine.

It is necessary to make calls in this order, because the data mapping must be done in a PL/I environment.

Main Storage Situation During Interlanguage Communication

To help with debugging, some of the main storage situations that can occur during interlanguage situations are shown in figures 13.9 through 13.11.

Options Assembler

The optimizing compiler provides a facility to simplify calling assembler language routines from PL/I. This consists of setting up an argument list that contains the addresses of all items passed rather than the addresses of locators.

When an entry point is declared as OPTIONS ASSEMBLER, argument lists passed to the entry point contain no locator addresses. The addresses of any areas, arrays, strings, or structures are passed directly in the parameter list. (For a call to a PL/I routine, the parameter list would contain the address of locators for these data types. This is because the

called routine might require information on the length or bounds of the data and this is accessible through the locator. See chapter 4 for details.)

The ASSEMBLER option does not provide facilities for automatically overriding PL/I interrupt handling, nor does it allow PL/I routines to be called from assembler language. If the programmer requires these facilities, he must provide the necessary code himself. The COBOL option without the INTER option provides complete facilities for calling, or being called by, assembler routines. However, its use involves the overhead of calls to the PL/I library interlanguage communication routines.

Full instructions on how to use PL/I with assembler language are given in the programmer's guide for this compiler.

Cobol Option in the Environment Attribute

A separate interlanguage communication facility offered by the compiler is the use of the COBOL option in file declarations. This option allows data sets created by COBOL programs to be read by PL/I programs and allows data sets to be created by PL/I programs in a format that is usable by COBOL programs. Interchange of data sets presents no problems, unless structures are used in the data set. If structures are used, their mapping may be different. (See above, under the heading "Handling Data Aggregate Arguments.") When structures are involved and the mapping is not known to be the same, both COBOL and PL/I structures are mapped, and compiled code transfers the data between structures immediately after reading the data for input, and immediately before writing the data for output.

During compilation, the compiler examines the record variable to see if any structures are involved. If no structures are involved, no further action need be taken. If structures are involved, a test is then made to see if the mapping of the structure or structures will be the same in COBOL and PL/I. If the compiler can determine that the mapping will be the same, then no action is required. If the compiler cannot determine that the mapping will be the same or if the structure is adjustable, the structure will be mapped in both the PL/I and the COBOL format. Adjustable structures will be mapped during execution by the resident library structure-mapping routines. Other structures will be mapped during compilation.

When re-formatting of data is necessary, the following actions take place when a record I/O statement involving a file with the COBOL option is executed.

INPUT:
: The data is read into a structure which has been mapped using the COBOL mapping algorithm and assigned to a PL/I mapped structure.

OUTPUT:
: Before the output takes place, the data in the PL/I structure is assigned to a structure mapped for COBOL. The output to the data set then takes place from the second structure.

The data assignment is carried out by compiled code in all circumstances.

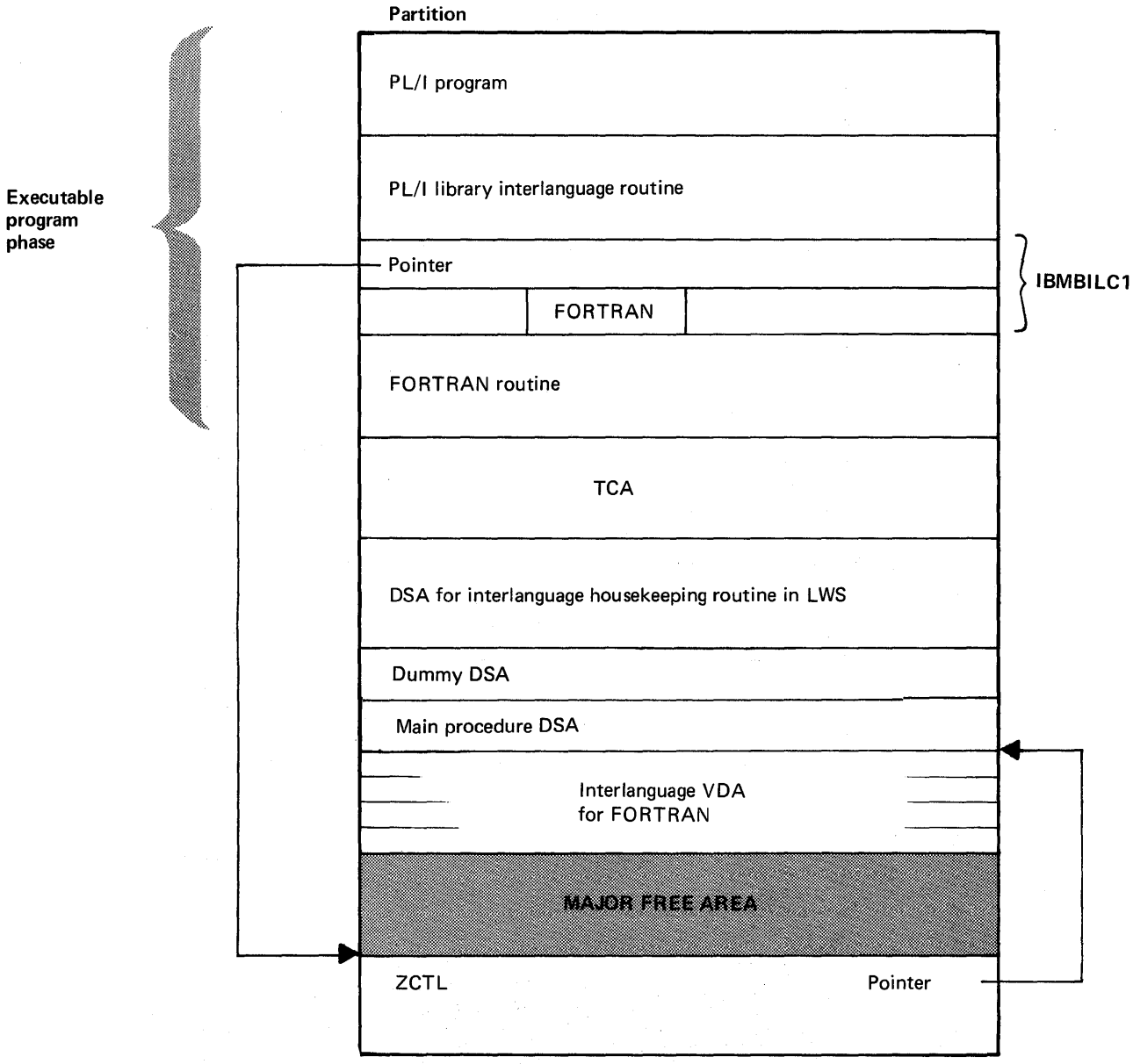


Figure 13.9. Main storage situation when PL/I main procedure calls FORTRAN

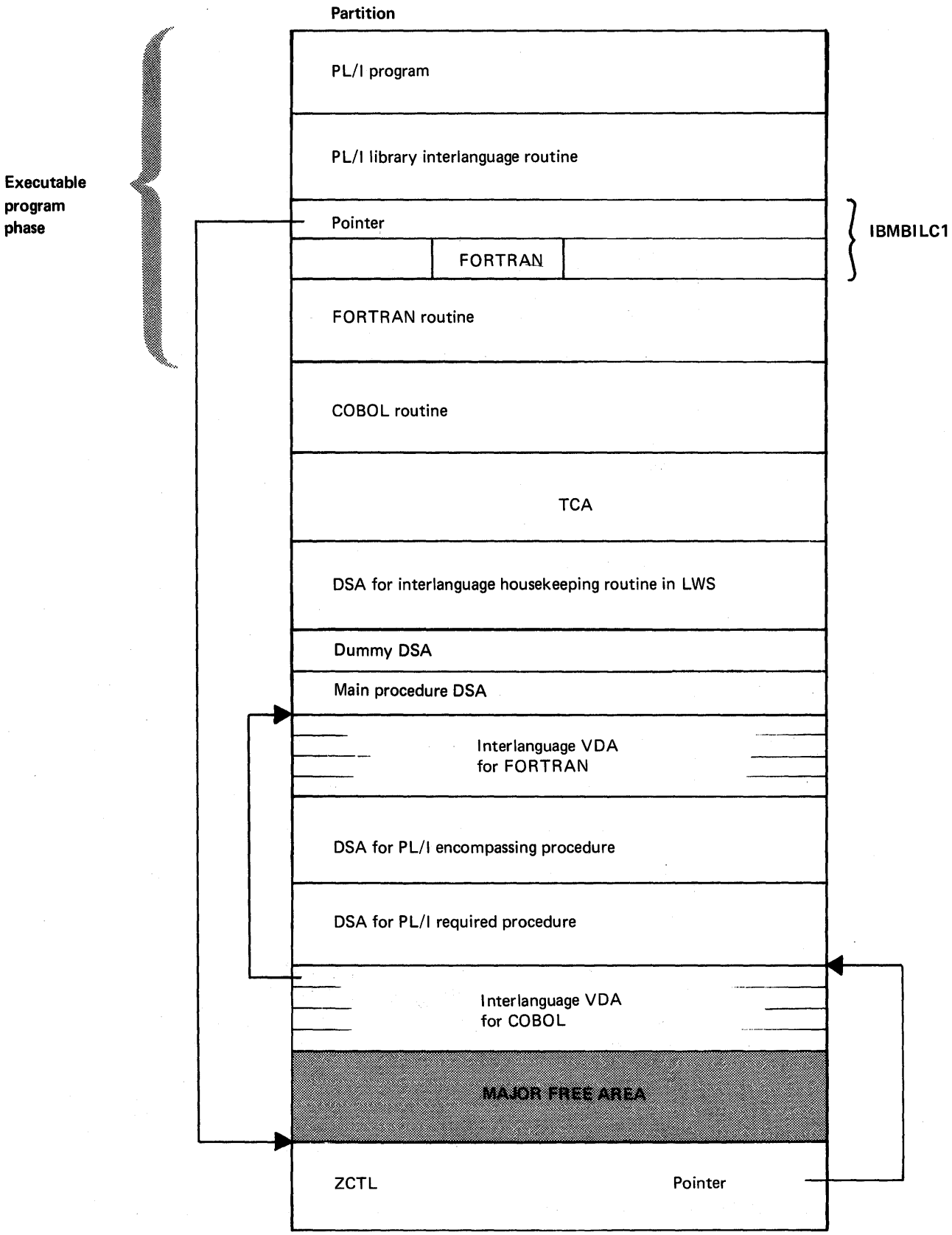


Figure 13.10. Main storage situation when PL/I main procedure calls FORTRAN, which in turn calls PL/I

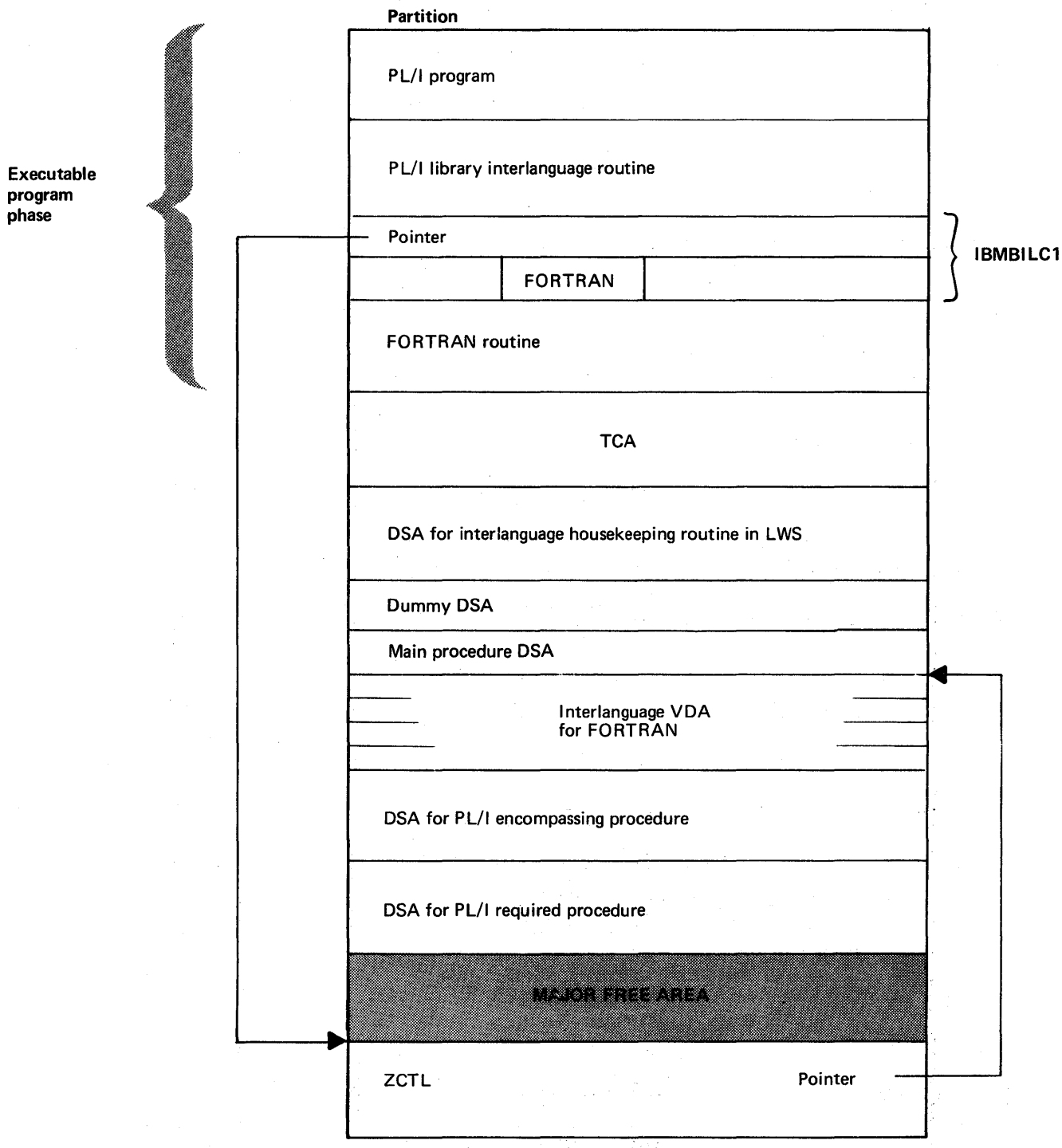


Figure 13.11. Main storage situation when PL/I main procedure calls FORTRAN, which calls PL/I, which calls COBOL

BEGINNING OF PARTITION

EXECUTABLE PROGRAM PHASE

Compiled code
All executable instructions generated by the compiler. Contents depend on source program.

Library subroutines
IBMDPIR - initialization routine (sets up TCA and other control blocks in program management area, then passes control to compiled code using address held in PLIMAIN)
IBMDERR - error and condition handling routine
IBMDPGR - storage management routine
Other routines, as necessary, for I/O conversions, etc.
LIOCS data management routines (if required)

Static storage (static internal and miscellaneous control sections)

PLISTART - Initial entry point.
Contains code to pass control to initialization routine IBMDPIR

PLIMAIN - Contains address of main procedure

Addresses of:
Compiled code entry points
External routines
Library routines
Controlled variables
Static external variables
External files
Label constants

R12

Control blocks:
Symbol table vector
Symbol tables
Parameter lists for library calls
Descriptors and locators
DEDs (data element descriptors)

File information:
FCB (file control block)
DTF (data management control block)
ENVB (environment control block)
OCB (open control block)
Filename

Diagnostic statement table
Non-qualified ONCBs
Constants used in program

Static variables less than 8 bytes
Static variables less than 2048 bytes
Other static variables

R12 → BOS ←

PROGRAM MANAGEMENT AREA

TCA

Flags	
Segment No.	BOS
Segment No.	EOS

Address of TRT table

Address of TCA appendage

Address of save area for IBMDPGR

Anchor for open-file chain

Address of DFB

Address of IBMBPGRD - VDA overflow routine

Address of IBMBPGRA - Get non-LIFO storage routine

Address of IBMBPGRB - Free non-LIFO storage routine

Address of IBMBPGRC - DSA overflow routine

Address of IBMDERR - Error and condition handling routine

TCA Appendage

TISA - Address of byte beyond ISA

TLFE - Address of last free area

Address of dummy DSA

Address of 'get LWS' routine

Address of load module chain

TERA - Address of interrupt handler

Dummy ONCA
(ON communications area)

Holds default values for PL/I condition built-in functions

TRT table

Translate-and-test table for IBMDERR, used in error handling to test for relevant on-cells

Diagnostic File Block (see appendix B)

Contains information relating to the use of SYSPRINT for the transmission of diagnostic messages

Save area for IBMDPGR

Used by storage management routines when new segment of storage is required

Dummy DSA (Dynamic storage area)

Contains DSA for initialization routine, backchain to calling routine's save area (if any), pointer to start of major free area (NAB), etc.

LWS (first allocation)

Flags	Offset to ONCA

Chainback

Register save area

Address of LWS

Segment No.	NAB

Workspace (56 bytes)

Flags	Offset to ONCA

Chainback

Register save area

Address of LWS

Segment No.	NAB

Workspace (56 bytes)

ONCA (see appendix B)

Chainback (to dummy ONCA)

Values of (or locators for) condition built-in functions

Caller's STXIT options

Save area for caller's STXIT program check options

LIFO STORAGE (Typ

Main procedure DSA

Chainback (to dummy DSA

Save area for main procedure's registers

Address of LWS

Segment No.	

Automatic variables and te
associated with the procedu
of dynamic ONCBs (ON co

R13 →

Subroutine DSA

Chainback

Register save area

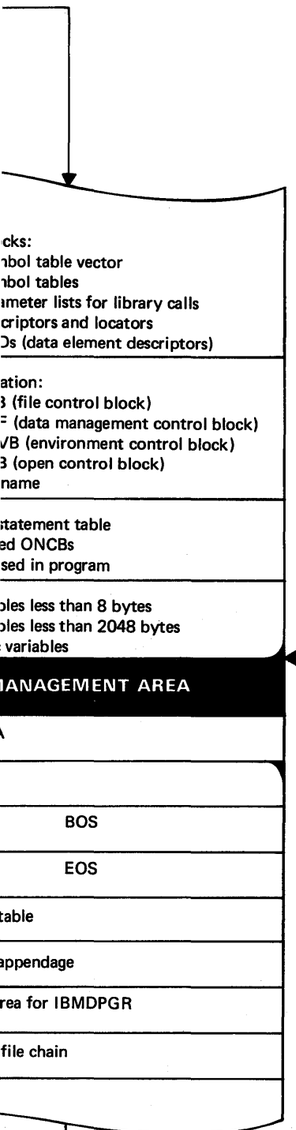
Address of LWS

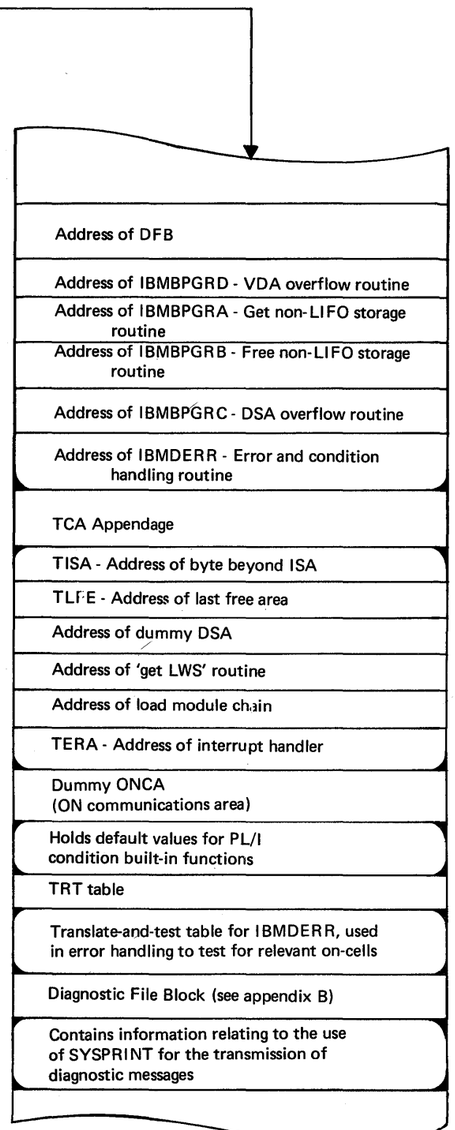
Segment No.	

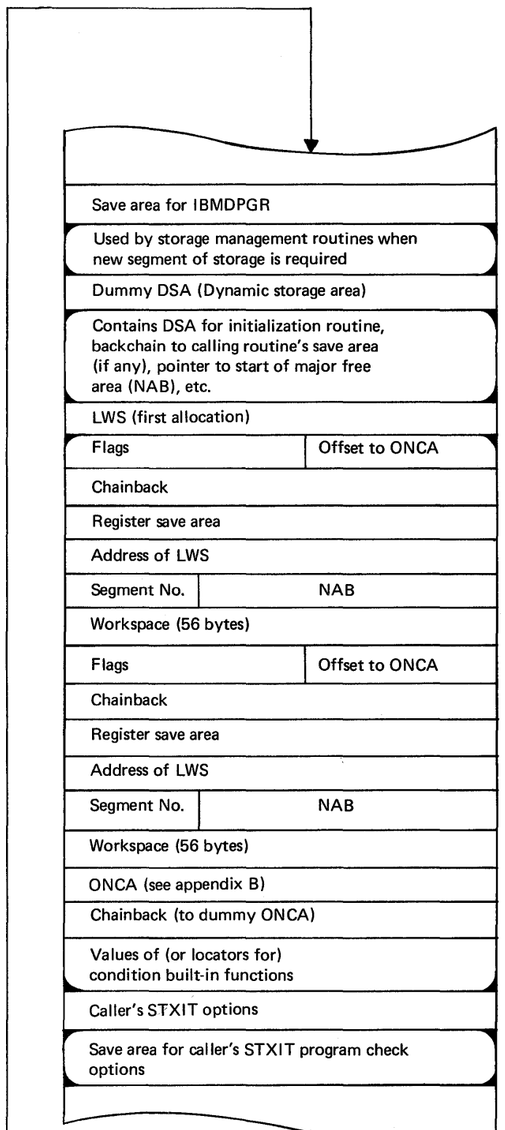
Automatic variables and ter
associated with subroutine,

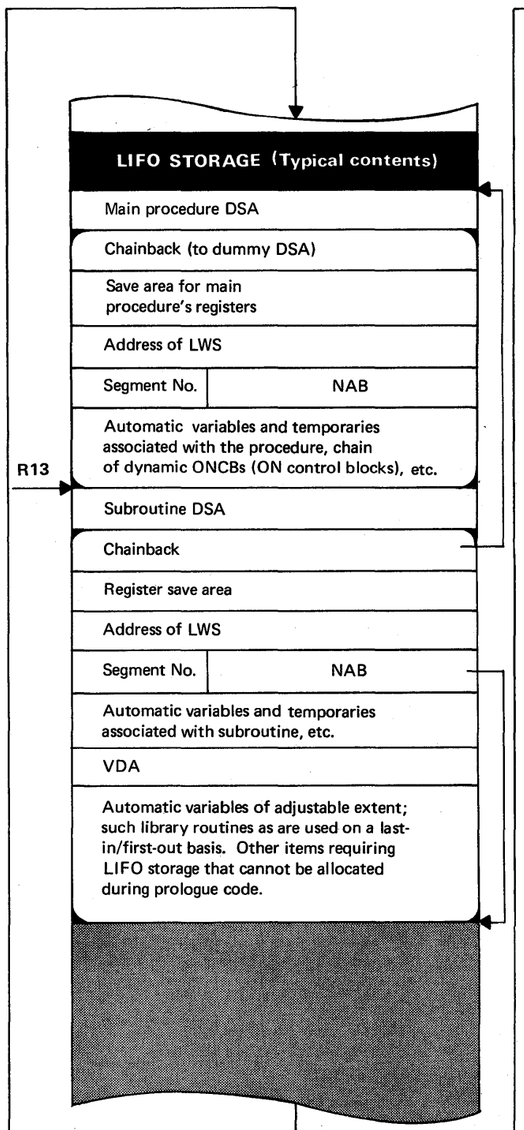
VDA

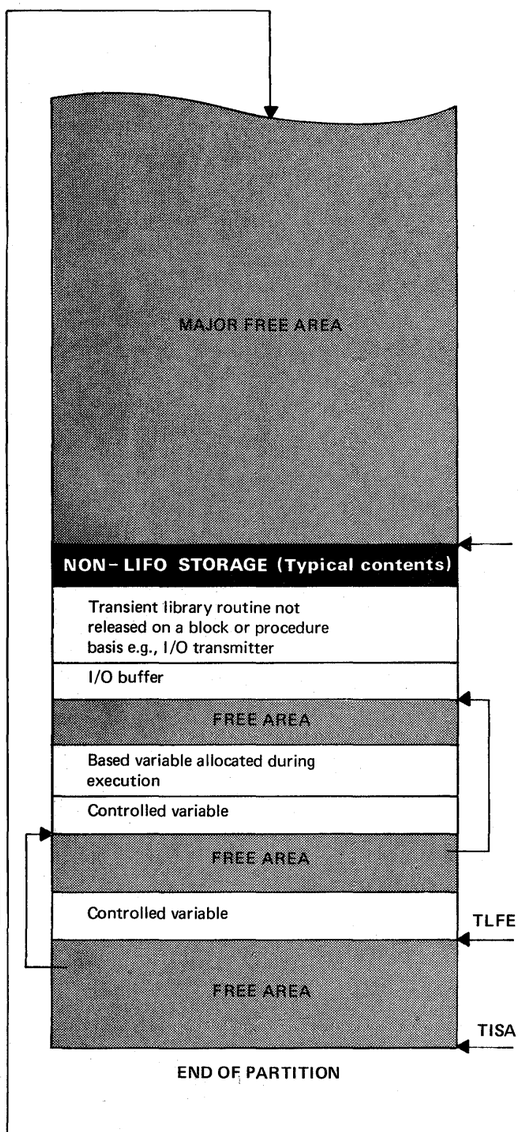
Automatic variables of adju
such library routines as are
in/first-out basis. Other iter
LIFO storage that cannot be
during prologue code.


cks:
nbol table vector
nbol tables
ımeter lists for library calls
criptors and locators
Ds (data element descriptors)
ation:
3 (file control block)
= (data management control block)
VB (environment control block)
3 (open control block)
name
statement table
ed ONCBs
sed in program
bles less than 8 bytes
bles less than 2048 bytes
variables
BOS
ANAGEMENT AREA
A
BOS
EOS
table
appendage
rea for IBMDPGR
file chain


Address of DFB
Address of IBMBPGRD - VDA overflow routine
Address of IBMBPGRA - Get non-LIFO storage routine
Address of IBMBPGRB - Free non-LIFO storage routine
Address of IBMBPGRC - DSA overflow routine
Address of IBMDERR - Error and condition handling routine
TCA Appendage
TISA - Address of byte beyond ISA
TLFE - Address of last free area
Address of dummy DSA
Address of 'get LWS' routine
Address of load module chain
TERA - Address of interrupt handler
Dummy ONCA (ON communications area)
Holds default values for PL/I condition built-in functions
TRT table
Translate-and-test table for IBMDERR, used in error handling to test for relevant on-cells
Diagnostic File Block (see appendix B)
Contains information relating to the use of SYSPRINT for the transmission of diagnostic messages


Save area for IBMDPGR
Used by storage management routines when new segment of storage is required
Dummy DSA (Dynamic storage area)
Contains DSA for initialization routine, backchain to calling routine's save area (if any), pointer to start of major free area (NAB), etc.
LWS (first allocation)
Flags
Offset to ONCA
Chainback
Register save area
Address of LWS
Segment No.
NAB
Workspace (56 bytes)
Flags
Offset to ONCA
Chainback
Register save area
Address of LWS
Segment No.
NAB
Workspace (56 bytes)
ONCA (see appendix B)
Chainback (to dummy ONCA)
Values of (or locators for) condition built-in functions
Caller's STXIT options
Save area for caller's STXIT program check options


LIFO STORAGE (Typical contents)
Main procedure DSA
Chainback (to dummy DSA)
Save area for main procedure's registers
Address of LWS
Segment No.
NAB
Automatic variables and temporaries associated with the procedure, chain of dynamic ONCBs (ON control blocks), etc.
R13
Subroutine DSA
Chainback
Register save area
Address of LWS
Segment No.
NAB
Automatic variables and temporaries associated with subroutine, etc.
VDA
Automatic variables of adjustable extent; such library routines as are used on a last-in/first-out basis. Other items requiring LIFO storage that cannot be allocated during prologue code.


MAJOR FREE AREA
NON-LIFO STORAGE (Typical contents)
Transient library routine not released on a block or procedure basis e.g., I/O transmitter
I/O buffer
FREE AREA
Based variable allocated during execution
Controlled variable
FREE AREA
Controlled variable
TLFE
FREE AREA
TISA
END OF PARTITION

Appendix B: Control Blocks

This appendix provides information on the format of the control blocks that may be used during the execution of a program compiled by the DOS PL/I Optimizing Compiler. Brief details of the function of each control block, together with when it is generated and where it can be located, are also given.

Except where explicitly stated all offsets from the start of a block are byte offsets and are given in hexadecimal notation.

Area Locator/descriptor

Function

Holds the address and length of the area variable for passing to other routines or for execution time reference if the area has an adjustable length.

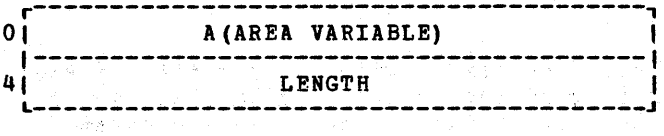
When Generated

As far as possible during compilation. If necessary completed during execution.

Where Held

Static internal control section.

How Addressed From an offset from register 3 known to compiled code.



Address of area variable is the address of the area variable control block.

Length is the total length including both the control block and the area variable.

AREA DESCRIPTOR

The area descriptor is the second word of the area locator descriptor. It is used in structure descriptors, when areas appear in structures, and in the controlled variable 'description' field when an area is controlled.

Area Variable Control Block

Function

Used to control storage allocation within the area variable.

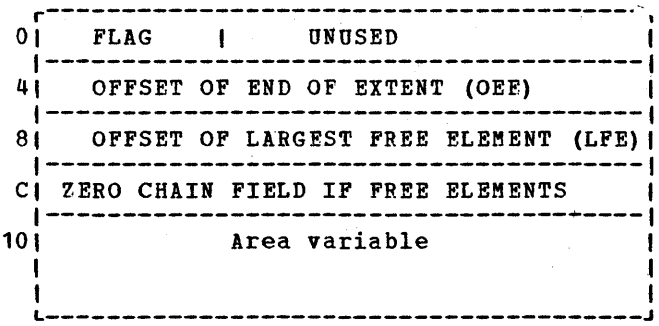
When Generated

When the area variable is initialized. This depends on the storage class of the area.

Where Held

How Addressed

As a variable dependant upon storage class. At the head of the area variable.



Note: If there are free elements in the area variable, they are headed by two words. The first word gives the length of the element, the second word gives the offset to the next smaller free element. If there is no smaller free element, the second word is set to zero.

Flag X'0' Area variable does not contain free elements.
X'1' Area variable does contain free elements.

Aggregate Descriptor Descriptor

Function

Contains information needed to map a structure or an array of structures during execution. Used for structures that contain adjustable extents or the REFER option. See chapter 4.

When Generated

As far as possible during compilation. Adjustable values are filled in during execution.

Where Held

Static internal control section.

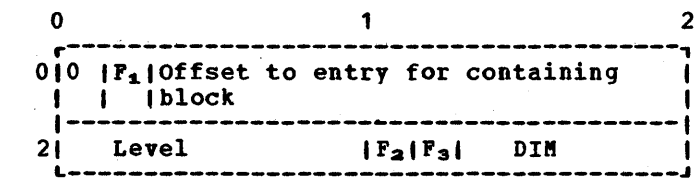
How Addressed

By an offset from register 3 known to compiled code

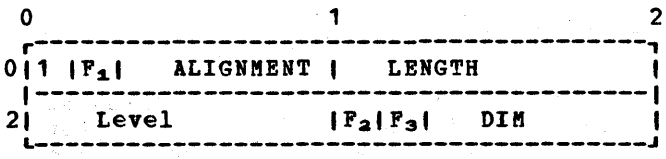
General Format

An aggregate descriptor descriptor consists of a series of fullword fields one for each structure element and one for each base element in the structure.

Structure Element



Base Element



where,

F_1 = '0'B Not last element in structure

= '1'B Last element in structure

F_2 = '0'B Not an AREA

= '1'B An AREA

F_3 = '0'B Not a BIT string

= '1'B BIT string

OFFSET = The offset within the aggregate descriptor descriptor to the entry for the containing structure. The offset is held in multiples of four bytes.

LEVEL = Logical level of identifier in structure

DIM = Real dimensionality of identifier

ALIGNMENT = Alignment stringency

Value(dec.)	Meaning
0	bit
7	byte
15	half-word
31	word
63	double-word

LENGTH = Length (in bytes) of data

LENGTH is set to 0 for strings and AREAs, whose length is held in descriptors

Aggregate Locator

Function

Used to pass the address of an array or structure and its associated descriptor to a called routine. Also to associate the aggregate with its descriptor during execution.

When Generated

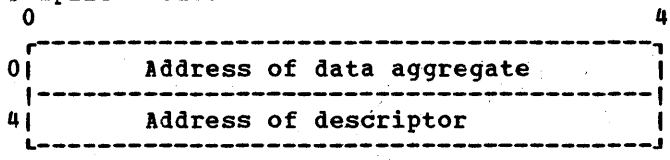
During compilation.

Where Held

Static internal control section.

How Addressed

By an offset from register 3 known to compiled code.



Array Descriptor

Function

Contains information about the extent of an array and the number of its dimensions. For arrays of area variables or strings, an area or string descriptor is attached to the array descriptor.

The array descriptor is used to pass information about an array to called routines, or to hold information about an array with adjustable extents.

When Generated

As far as possible during compilation. If the array has adjustable extents, it is completed during execution when the values are known.

Arrays of structures make use of structure descriptors to hold similar information.

Where Held

Static internal control section.

How Addressed

By an offset from register 3 known to compiled code

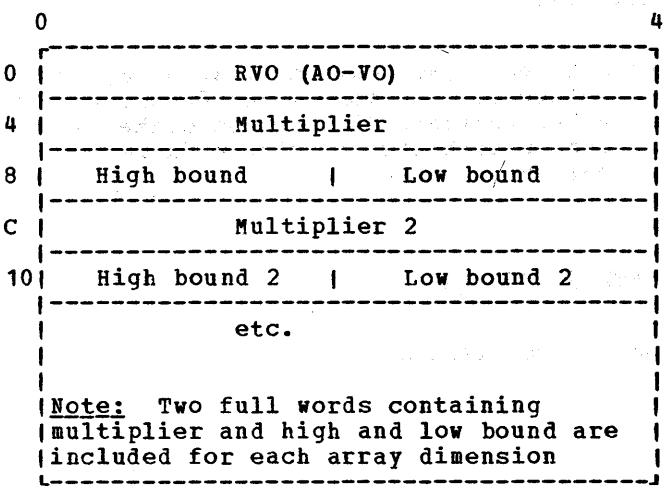
Arrays of Strings or Areas

For arrays of strings or areas, the descriptors are completed by string or area descriptors concatenated to the array descriptor. String and area descriptors are the second word of string and area descriptor/locator pairs.

For bit string arrays, the bit offset from the byte address is held in the string descriptor.

General Format

The first word in the array descriptor is the RVO (relative virtual origin). This is followed by two words for each dimension of the array, containing the multiplier and high and low bound for each dimension.



RVO = Relative virtual origin, the distance between the virtual origin (VO) and the byte actual origin (AO). Virtual origin is the point at which the element in the array whose subscripts are all zeros is, or would be, held. Actual origin is the byte address of the first element in the array.

RVO is held as a bit value for arrays of unaligned non-varying bit strings, but otherwise as a byte value.

High bound: The highest subscript in the dimension.

Low bound: The lowest subscript in the dimension.

Multiplier: The multiplier is the offset between any two elements marked by the change of subscript number in the dimension.

For example for the array DATA(10,10), the multiplier for the first dimension is the offset between DATA(1,1) and DATA(2,1) etc. The multiplier for the second dimension is the offset between DATA(1,1) and DATA(1,2). The offset is measured from the start of the one element to the start of the next.

Multipliers are byte values except for unaligned non-varying bit string arrays, in which case they are bit values.

Controlled Variable Block

Function

To hold information about the controlled variable.

When Generated

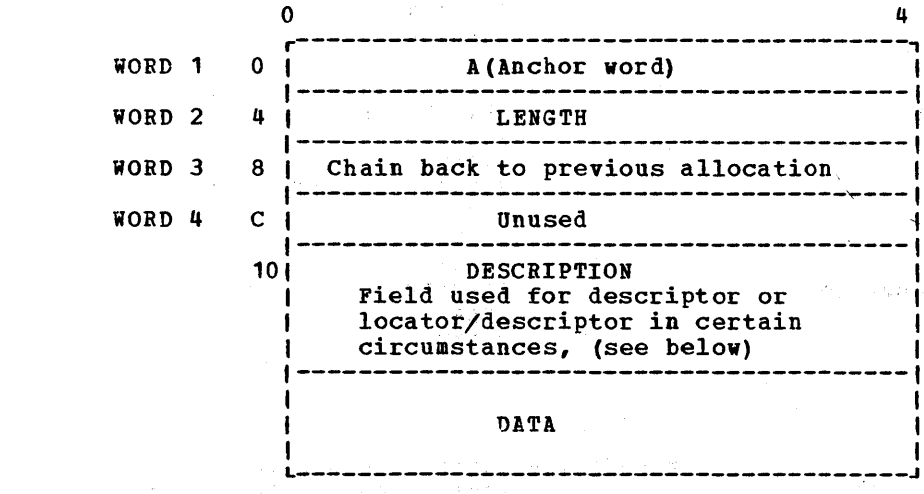
When the variable is allocated.

Where Held

At the head of the controlled variable.

How Addressed

The latest allocation is addressed from an anchor word which is held in static internal storage for internal variables and in a separate control section for external variables.



LENGTH: Length of the total allocation including the 4 words of the heading.

CHAIN BACK: Address of word 5 of previous allocation, set to zero if first allocation.

DESCRIPTION

If the item is one that requires a descriptor/locator or a locator, this is placed at the head of the data. If the item is a structure or array and the extents are unknown at compile time, the descriptor will also be placed before the data. class.

Thus for:

STRINGS and AREAS, the controlled variable is headed by a locator/descriptor.

STRUCTURES and ARRAYS, the controlled variable is headed by a locator.

STRUCTURES and ARRAYS with ADJUSTABLE EXTENTS, the controlled variable is headed by a locator followed by a descriptor.

ALL OTHER DATA, the description field is not used and the data itself starts at offset X'10' (16)

Data Element Descriptor (DED)

Function

Used to pass description of data elements to library conversion and stream I/O routines.

When Generated

During compilation.

Where Held

Static internal control section.

How Addressed

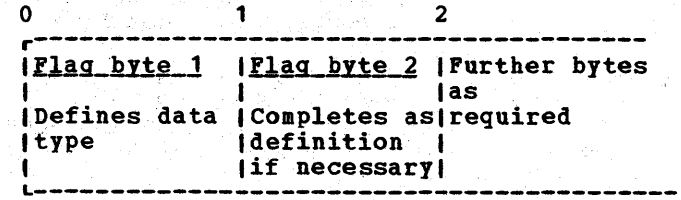
By an offset from register 3 known to compiled code.

Format of DEDs

All DEDs are headed by two bytes that indicate the data type. These two bytes are followed by as many bytes as are required to complete the description of the data.

For arithmetic items, DEDs are completed by such items as scale and precision. For pictured items, a representation of the picture is included in internal form.

General Format



Flag Byte 1

Hex Value	Data Type
00	FIXED BINARY
04	FIXED DECIMAL
08	FLOAT
0C	Free decimal (an internal form)
10	FIXED PICTURE BINARY
14	FIXED PICTURE DECIMAL
18	FLOAT PICTURE BINARY
1C	FLOAT PICTURE DECIMAL
20	non-VARYING CHARACTER
24	non-VARYING BIT
28	VARYING CHARACTER
2C	VARYING BIT
30	CHARACTER PICTURE
40	BINARY constant
44	DECIMAL constant
48	BIT constant
50	F/E Format
54	P Format (arithmetic)
58	A/B/P Format (character)
5C	C Format
60	X Format
64	COL Format
68	SKIP Format
6C	LINE Format
70	PAGE Format
80	LABEL
84	ENTRY
88	AREA
8C	TASK
90	OFFSET
94	POINTER
98	FILE
9C	EVENT

Flag Byte 2

Bits 0&1 = '00'B A-format item
'01'B B-format item '10'B character picture format item

Bit 2 = '0'B fixed constant
'1'B float constant

Bit 3 = '0'B not extended float
'1'B extended float

Bit 4 = '0'B F-format/fixed picture
'1'B E-format/float picture

Bit 5 = '0'B declared binary
'1'B declared decimal

If bits 4 and 5 = '11'B then DED is for character

Bit 6 = '0'B short precision
'1'B long precision

Bit 7 = '0'B real or length specified (A or B format) or unaligned bit string
'1'B complex (also set if E, F, or P in C-format) or no length specified (A or B format) or aligned bit string

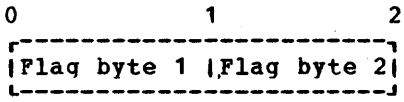
All bits for which neither value is defined are set to '0'B

Internal codes for pictures

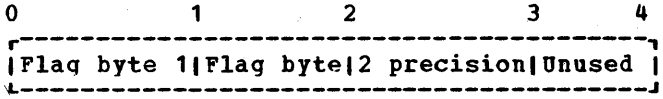
Code	Picture	Code	Picture
00	9	48	- (t)
04	Y	4C	- (d)
08	Z	50	- (s)
0C	*	54	$ (t)
10	E	58	$ (d)
14	K	5C	$ (s)
18	T	60	/ (t)
1C	I	64	/ (d)
20	R	68	/ (s)
24	CR	6C	. (t)
28	DB	70	. (d)
2C	B	74	. (s)
30	S (t)	78	, (t)
34	s (d)	7C	, (d)
38	S (s)	80	, (s)
3C	+	84	V
40	+ (d)		
44	+ (s)		

(t) = terminal
(d) = drifting
(s) = static

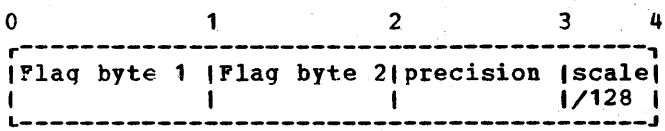
DED for STRING Data



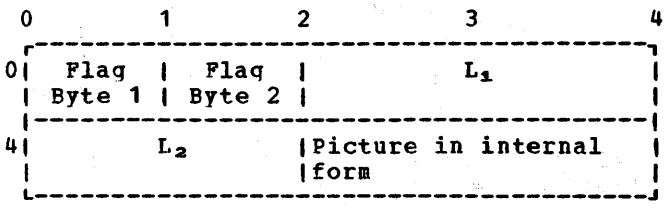
DED for FLOAT Data



DED for FIXED Data



DED for PICTURE STRING Data



Flag byte 1 = Hex 30

The internal code for string pictures is as follows:

Code	Picture (hex)
A	00
9	04
/	08
.	0C
,	10
*	14
B	18
x	1C

L_1 = length of field with insertion characters

L_2 = length of field without insertion characters

DED for PICTURE DECIMAL Arithmetic Data

Flag byte 1 = Hex 14 or 1C

Flag Byte 3 (describes the mantissa subfield)

Bit 0 = reserved; must be set to '0'B

Bit 1 = '1'B drifting S in subfield
= '0'B no drifting S in subfield

Bit 2 = '1'B drifting + in subfield
= '0'B no drifting + in subfield

Bit 3 = '1'B drifting - in subfield
= '0'B no drifting - in subfield

Bit 4 = '1'B drifting $ in subfield
= '0'B no drifting $ in subfield

Bit 5 = '1'B total suppression in subfield
= '0'B no total suppression in subfield

Bit 6 = '1'B * in subfield
= '0'B no * in subfield

Bit 7 = reserved; must be set to '0'B

Flag Byte 4 (describes the exponent subfield)

Same format as Flag Byte 3.

Note: After E or K, the next byte contains the number of digits in the exponent.

Scale Factor

The scale factor of a picture DED is the number of digit positions after the 'V' (0 if there is no 'V') added to the number in the F specification, if any.

Rule for setting bit 5 in Flag Bytes 3 and 4

Bit 5 is set if no 9, Y, T, I, or R is present. This applies before any Z, S, etc. has been translated to a 9.

Rules for translating pictures into encoded pictures

1. Characters 9, Y, E, K, T, I, R, CR, DB, B, and V are translated directly.
2. Characters Z and * are translated directly if they do not follow a V. If either follows a V, it is translated into the code for character 9.
3. An S, +, -, or $ is translated to a static S, +, -, or $ if it is the only one of its kind in the subfield.
4. If more than one S appears in a subfield, the S's are translated into drifting S's.

Except when:

   a. It appears immediately before a Y, 9, V, T, I or R. In this case it is translated into the code for a terminal S.

   b. It appears anywhere after a V. In this case it is translated into the code for a 9.

   The same rule applies for the +, -, or $.

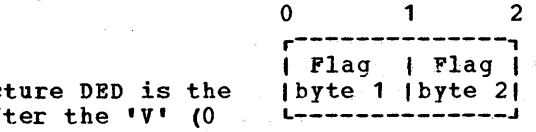
5. A "/", a ",", or a "." is treated as drifting, if:

   a. It is in a subfield containing either one or more Z or asterisk, or more than one +s, -s, or $.

and if:

   b. It is not immediately preceding a Y, 9, V, T, I, or R. In this case it is translated into terminal form.

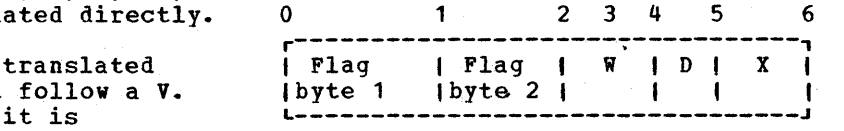
DED for Program Control Data



FORMAT DEDS - FEDS

For meaning of flag bytes see above under Data Element Descriptors.

DED for F and E Format Items (FED)



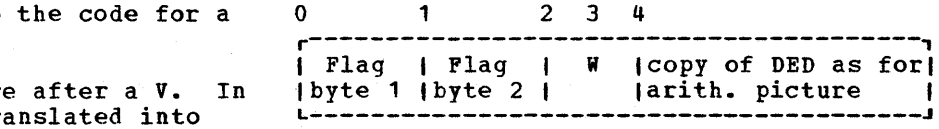
Flag byte 1 = Hex 50

W = total length of the format field

D = number of decimal places

X = precision + 128 for F-format number of significant figures for F-format

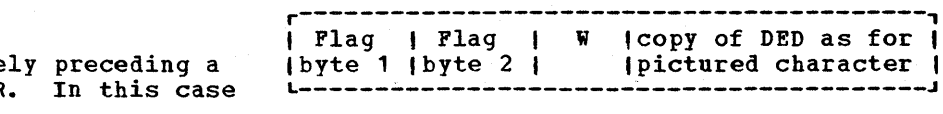
DED for PICTURE Format Arithmetic Items (FED)



Flag byte 1 = Hex 54

W = total length of the format field

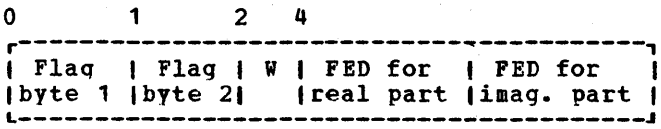
DED for PICTURE Format Character Items (FED)



Flag byte 1 = Hex 58

W = total length of the format field

DED for C Format Items (FED)

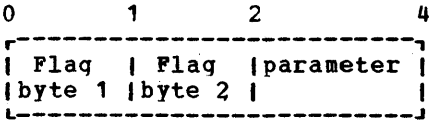


Flag byte 1 = Hex 5C

Note: The complex bit (bit 7) in flag byte 2 is set in both the real part and the imaginary part FED.

W = total length of the format field

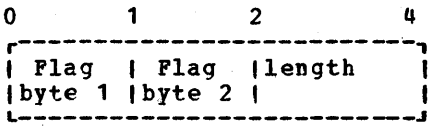
DED for Control Format Items (FED)



Flag byte 1 = Hex 60, 64, 68, 6C or 70

Parameter = length of item (X format)
column number (COL format)
number of lines to skip (SKIP format)
line number (LINE format)
omitted for PAGE format

DED for STRING Format Items (FED)



Flag byte 1 = Hex 58

The difference between A, B, and P (character) formats is given by bits 0 and 1 of flag byte 2. The length field may be omitted for A and B format items.

Diagnostic File Block (DFB)

Function

Holds information used by the error message routines.

When Generated

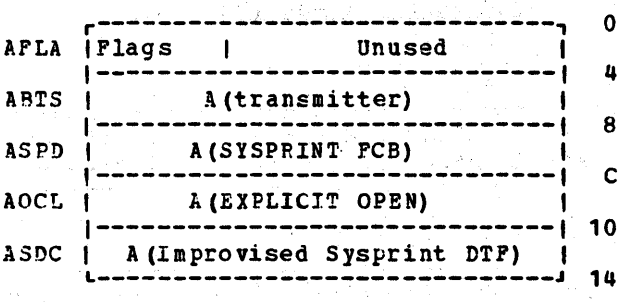
During program initialization.

Where Held

Program managment area.

How Addressed

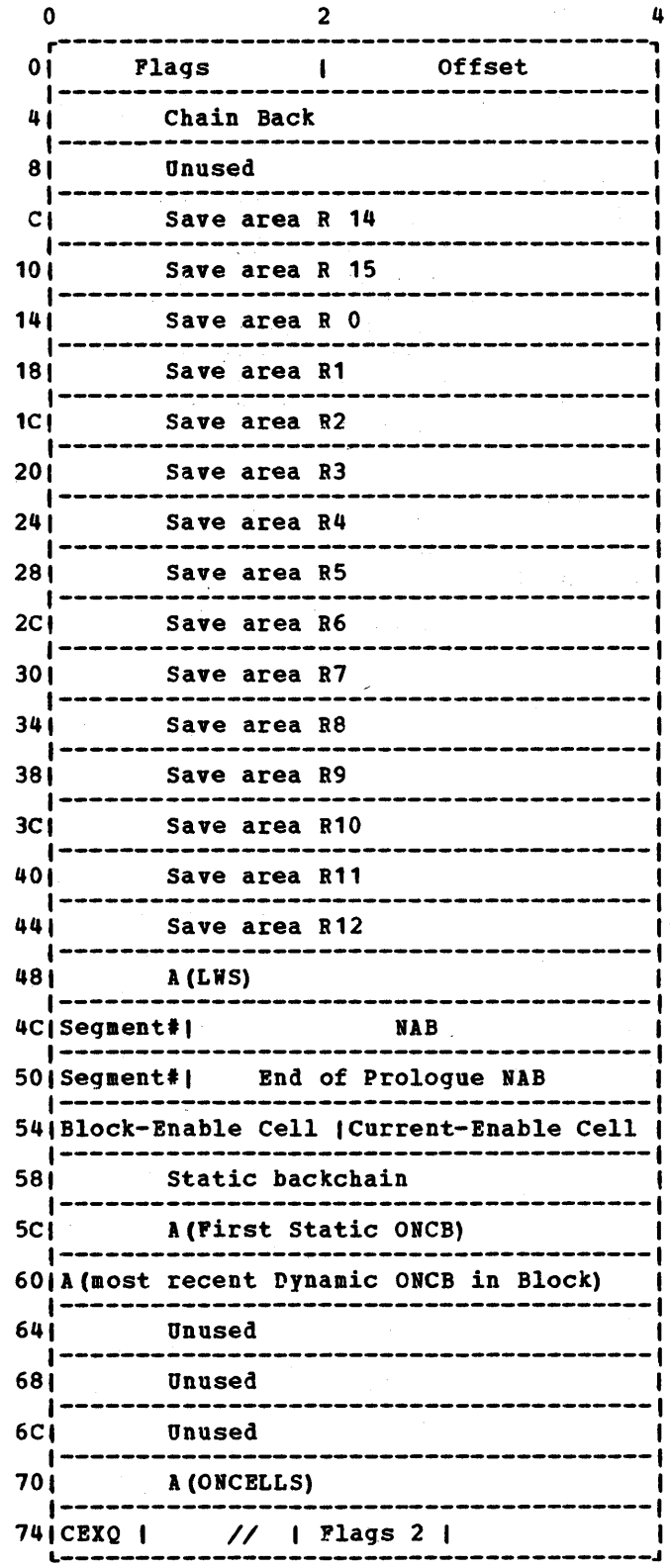
From X '40' in the ICA



AFLA - Flags

AWTO Bit 0 = 1 Messages going to operator's console
ASNO Bit 1 = always 0
ASCO Bit 2 = 1 SYSPRINT cannot be opened or open with unsuitable attributes.
AFPF Bit 3 = 1 Force page

Dynamic Storage Area (DSA)



Function

Holds housekeeping information, automatic variables, and temporaries for each block.

When Generated

During execution. Allocated by prologue code every time a new block is entered.

Where Held

In the LIFO storage stack. Certain library routines have their DSAs in library workspace (LWS). See below

How Addressed

Current DSA addressed from register 13. Chainback to previous DSAs from offset X '4'.

Flags

Bit 0 = 0 DSA in LWS
1 DSA

Bit 1 = 0 No ON Cells
1 ON cells

Bit 2 = 0 No Dynamic ONCBs
1 Dynamic ONCBs

Bit 3 Always set to zero.

Bits 4 and 5
= 00 Procedure DSA
01 Begin DSA
10 Library DSA
11 On DSA

Bit 6 = 0 Not a dummy DSA
1 Dummy DSA

Bit 7 = 0 Flags 2 invalid
1 Flags 2 valid

Bit 8 = always zero

Bit 9 = 0 Do not restore NAB on GOTO
1 Restore NAB

Bit 10 = 0 Do not restore Current-enable on GOTO
1 Restore current-enable cells

Bit 11 = 0 Callee cannot use this DSA
1 Callee can use this DSA

Bit 12 = 0 Not an EXIT DSA
1 EXIT DSA

Bit 13 = 0 No statement # table
1 Statement # table available

Bit 14 = always zero

Offset

If the DSA is in LWS, offset is the offset of the ONCA. Otherwise, this field is not used.

CEXQ

Save area for flag byte 1 of the TCA. Used if DSA is an exit DSA.

Flags_2

Bit 0 = 1 Last PL/I DSA

Bit 1 = 1 Ignore DSA for SNAP

Bit 3 = 1 Inter-language DSA after interrupt in FORTRAN or COBOL

Entry Data Control Block (Entry Variable)

Function

Holds the addresses of the data item and its DSA.

When Generated

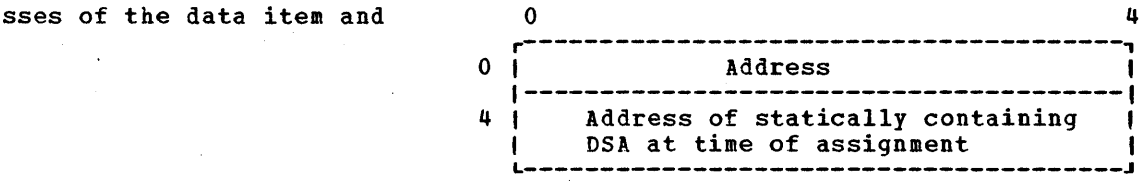
When the variable is allocated.

Where Held

Depends on the storage class of the data item.

How Addressed

As a variable, dependant on storage class.



Word 1: bit 0 = 0 Address of entry
= 1 Address of location containing 8-char. EBCDIC name of entry point

Word 2: bit 0 always = 0

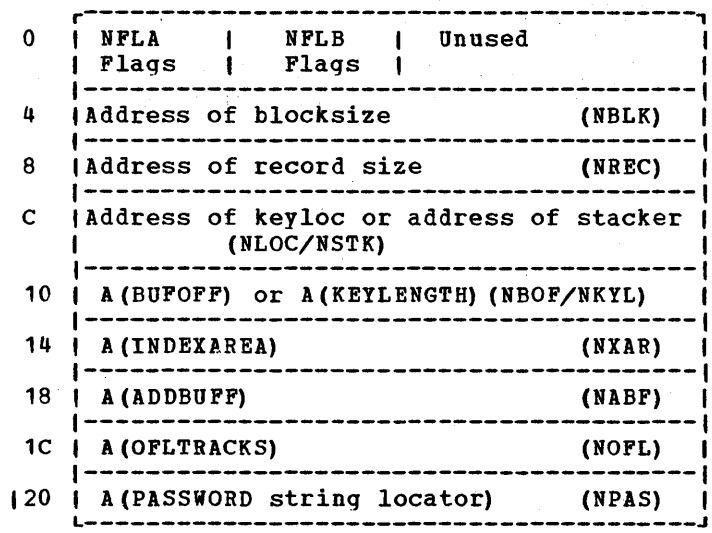
Environment Block (ENVB)

Function

Holds addressed of information declared in the environment option

When Generated

During compilation



NFLA Flags

Bit 0 when set BLOCKSIZE field valid.
1 when set RECORDSIZE field valid.
2 when set KEYLOC field valid.
3 when set BUFOFF field valid
4 when set KEYLENGTH field valid.
5 when set INDEX AREA field valid.
6 when set ADDBUF field valid.
7 when set OFLTRACKS valid.

NFLB flags

Bit 0 Function (R) (read)
Bit 1 Function (P) (punch)
Bit 2 Function (W) (write)
Bit 3 reserved
Bit 4 when set STACKER field valid
Bit 5 reserved
Bit 6 when set PASSWORD field valid
Bit 7 reserved

Where Held

In the static external control section for the file if file is external. Otherwise in the internal static control section.

How Addressed

By an address contained in the FCB of the file.

Addresses

The addresses held are the locations where compiled code will have placed the correct values for the current environment.

Event Table (EVTAB)

Function

Used by WAIT module as workspace and to provide status information on associated event.

When Generated

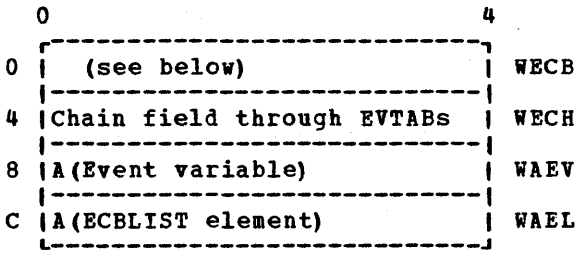
During execution.

Where Held

In LIFO storage.

How Addressed

Address known to WAIT module



WECB Bit 0 set when event is complete Bits 1-7 Not used in this implementation

Event Variable Control Block

Function

To hold information about the operation with which the EVENT has been associated.

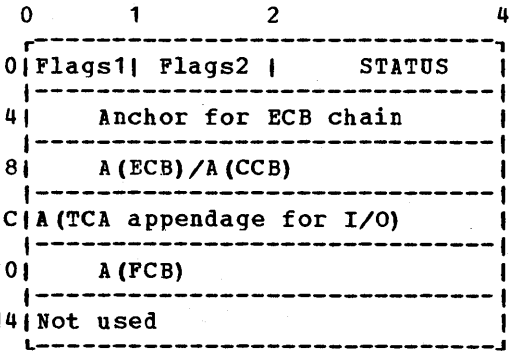
When Generated

Depends on the storage class of the event variable.

Where Held

Depends on the storage class of event variable.

How Addressed As a variable, dependand upon storage class.



Flags 1

Bit 0 =0 Incomplete
1 Complete

Bit 1 =0 Inactive
1 Active

Bit 2 =0 Not an I/O EVENT
1 I/O EVENT

Bit 3 =0 Not a DISPLAY EVENT
1 DISPLAY EVENT

Bit 4 =0 EV has not caused on-unit entry
1 EV has caused entry to an on-unit

Bit 7 =always zero

Flags 2

Bit 0 =0 No chain of ECBs
1 Chain of ECBs exists

Bit 1 =0 Not a dummy EVENT
1 Dummy EVENT

File Control Block (FCB)

Function

Used to access all file information. Contains addresses of the ENVB, DTF, ACB, filename, etc.

When Generated

As far as possible during compilation. Completed by the open routines during execution.

Where Held

In the static internal control section for internal files.
In the associated file control section for external files.

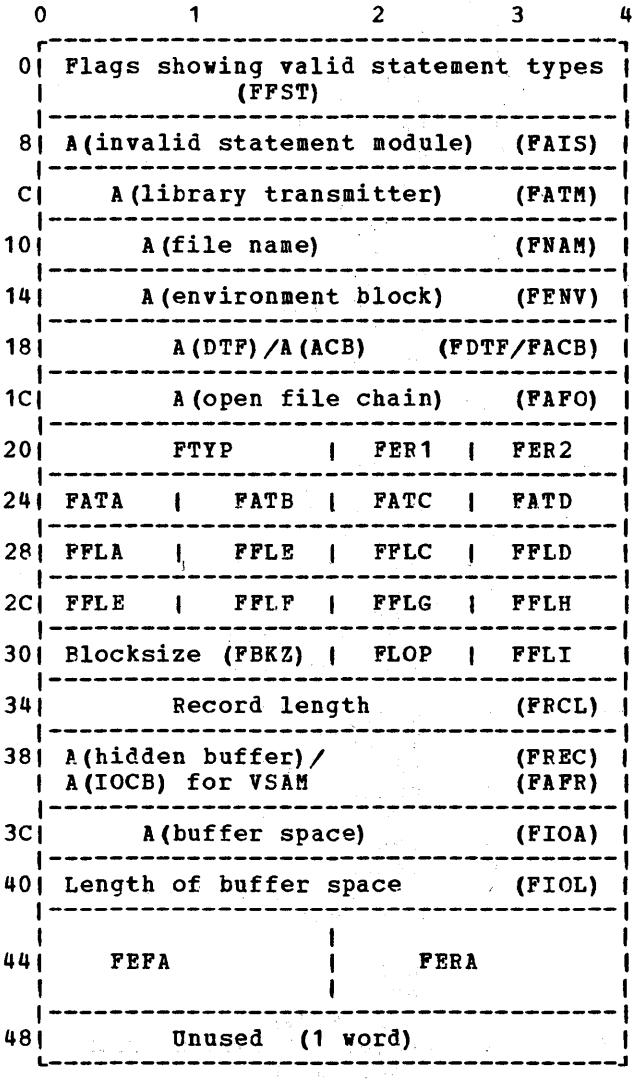
How Addressed By an offset from register 3 if internal. Address filled in by linkage editor if external.

Common Section

FFST Flags indicating types of statement (8 bytes)

Bit number	Statement + options
0	READ SET
1	READ SET KEYTO
2	READ SET KEY
3	READ INTO
4	READ INTO KEYTO
5	READ INTO KEY
6	READ INTO KEY NOLOCK
7	READ IGNORE
8	READ INTO EVENT
9	READ INTO KEYTO EVENT
10	READ INTO KEY EVENT
11	READ INTO KEY NOLOCK EVENT
12	READ IGNORE EVENT
13	WRITE FROM
14	WRITE FROM KEYFROM
15	WRITE FROM EVENT
16	WRITE FROM KEYFROM EVENT
17	REWRITE
18	REWRITE FROM
19	REWRITE FROM KEY
20	REWRITE FROM EVENT
21	REWRITE FROM KEY EVENT
22	LOCATE SET
23	LOCATE SET KEYFROM
24	DELETE
25	DELETE KEY
26	DELETE EVENT
27	DELETE KEY EVENT
28	UNLOCK KEY
29-63	Reserved

Common Section



FTYP 6th and 7th characters of library transmitter name

FER1 and FER2 Error Flags

FATA-FATD Flags showing attributes allowable with file types, and other file usage information.

	Bit	Attribute
FATA	0	(Open SYSPRINT for error

		message)
	1	(SYSPRINT)
	2	unused
	3	(String operation)
	4	unused
	5	DISPLAY
	6	RECORD
	7	STREAM
FATB	0	BACKWARDS
	1	UPDATE
	2	OUTPUT
	3	INPUT
	4	unused
	5	unused
	6	DIRECT
	7	SEQUENTIAL
FATC	0	unused
	1	unused
	2	unused
	3	PRINT
	4	unused
	5	KEYED
	6	UNBUFFERED
	7	BUFFERED
FATD	all	unused

FFLA-FFLH Flags Bit

FFLA	0	F-format
	1	V-format
	2	U-format
	3	Spanned
	4	Blocked
	5	unused
	6	unused
	7	Key in record variable KEYLOC
FFLB	0	CONSECUTIVE
	1	INDEXED
	2	REGIONAL(1)
	3	unused
	4	REGIONAL(3)
	5	unused
	6	unused
	7	other organization (see FFLI)
FFLC	0	LEAVE
	1	UNLOAD
	2	BUFFERS(2)
	3	Permanent output error (mag.tape)
	4	stacker(2)
	5	associated file
	6	3881 device
	7	unused
FFLD	0	Transmitter deblocking
	1	ERET macro
		permitted
	2	DTF completed by compiler
	3	buffer length calculated by compiler
	4	ENDFILE module loaded
	5	unused
	6	error module loaded
	7	GENKEY
FFLE	0	I/O error
	1	permanent input error or non-scalar varying, length set
	2	permanent output error
	3	end of file
	4	hidden buffer in use
	5	move required
	6	non-SCALARVARYING
	7	operation checked
FFLF	0	previous READ
	1	previous READ SET
	2	previous LOCATE
	3	previous REWRITE
	4	previous OPEN
	5	close in progress
	6	implicit close
	7	outstanding event or previous open (VSAM resume load)
FFLG	0	Endpage
	1	end of extents
	2	COPY option active
	3	CONSEC UNBUFF
	4	unused
	5	Synad entered
	6	Newly opened print file
	7	File open
FFLH	0	In-line I/O
	1	In-line locate
	2	unused
	3	unused
	4	unused
	5	Blanks at end of record
	6	unused
	7	unused
FLOP	7th	character of OPEN module name
FFLI	0	VSAM
	1-7	reserved

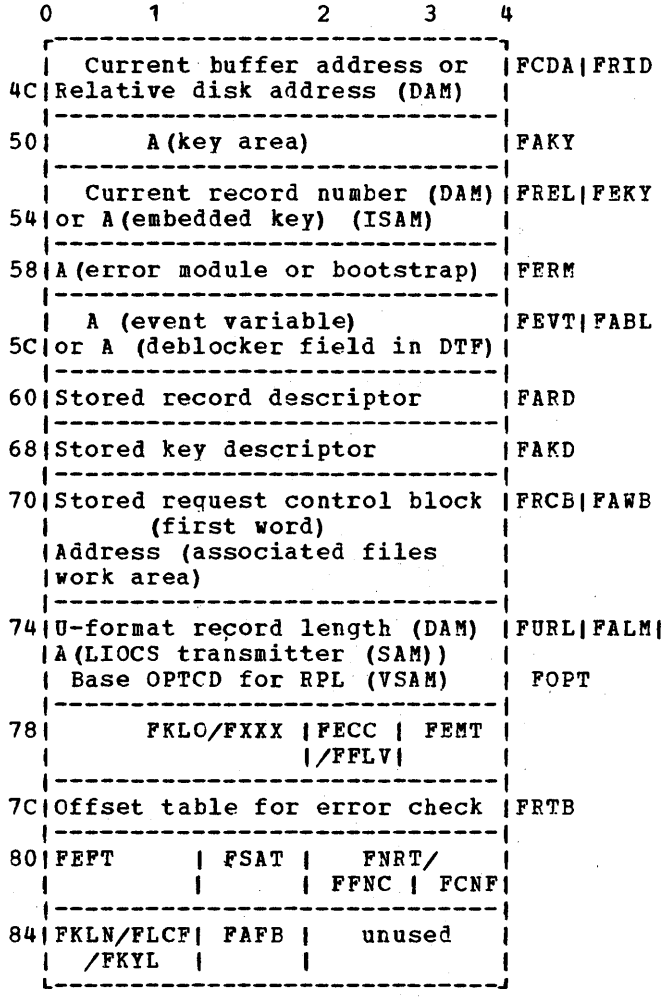
FEFA offset of EOFADDR in DTF

FERA offset of ERROPT address in DTF

The common section is followed by either the RECORD or STREAM sections.

Record I/O Section

Offsets are from start of the FCB.



FKLO KEYLOC-1

FXXX Error bytes for DAM

FECC 2 for BACKWARDS (12 for FORWARDS) (Mag Tape only)

FFLV 0 KSDS 1 ESDS 2-7 reserved

FEMT 7th char of error module name

FEFT 7th char of endfile module name

FSAT saved attributes (consec unbuff)

FNRT number of records/track (DAM)

FFNC Associated file byte

FCNF Associated file conflicting operations flags

FKLN Keylength-1

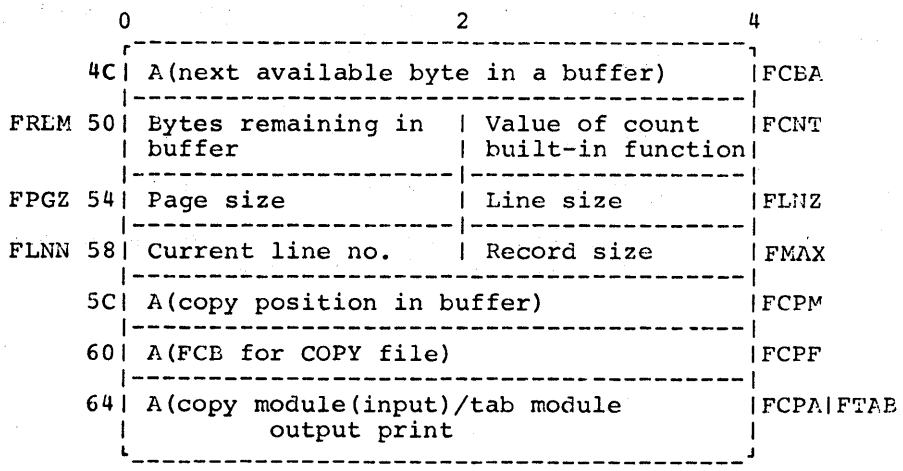
FLCT Number of lines left on card

FKYL Keylength (VSAM only)

FAFB Associated file work byte

Stream I/O Section

Offsets are from the start of the FCB.



Flow Statement Table

Function
Used to implement the compiler FLOW option. Holds the last 'n' statement number pairs and the last 'm' procedure executed. ('n' and 'm' are programmer defined.)

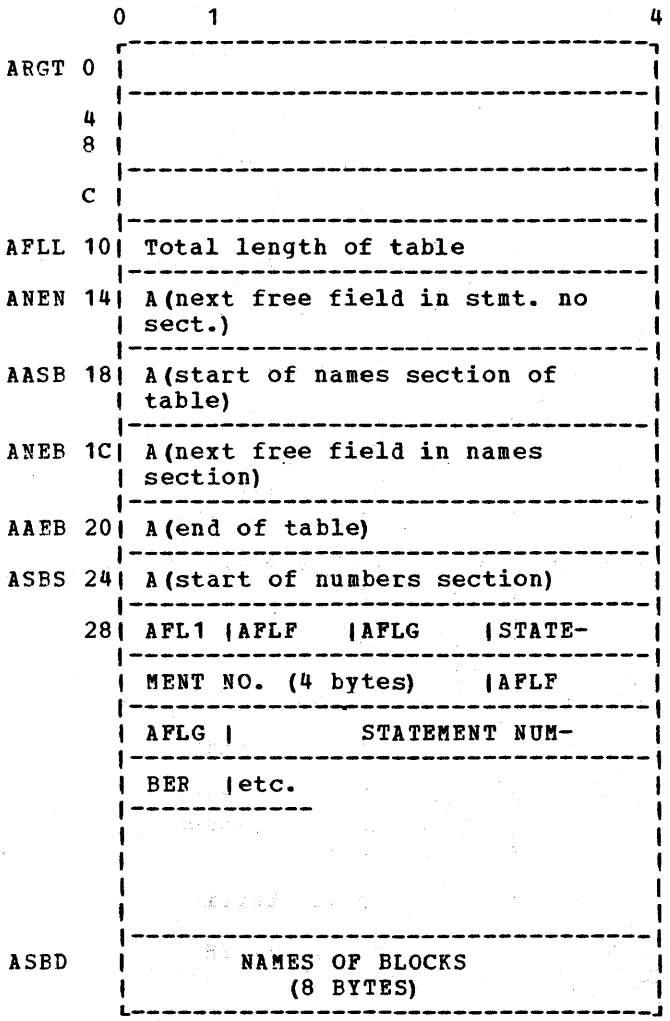
When Generated
initialization if the FLOW option has been specified. The table is continually updated as the prgoram is executed.

Where Held

Static internal control section.

How Addressed

From offset X '4C' in the TCA.



AFLG - Flags

unused

AFL1 - Flags

ANON Bit 0 No statement numbers required
AFLI Bit 1 last entry was branch-in
AILF Bit 2 unused
AINT Bit 3 interrupt not recorded
AGOT Bit 4 GOTO out of block
Bits 5-7 unused

AFLF - Flags

ATBI Bit 0 Branch-in entry
ABCD Bit 1 BCD form for this entry
Bits 2-7 unused

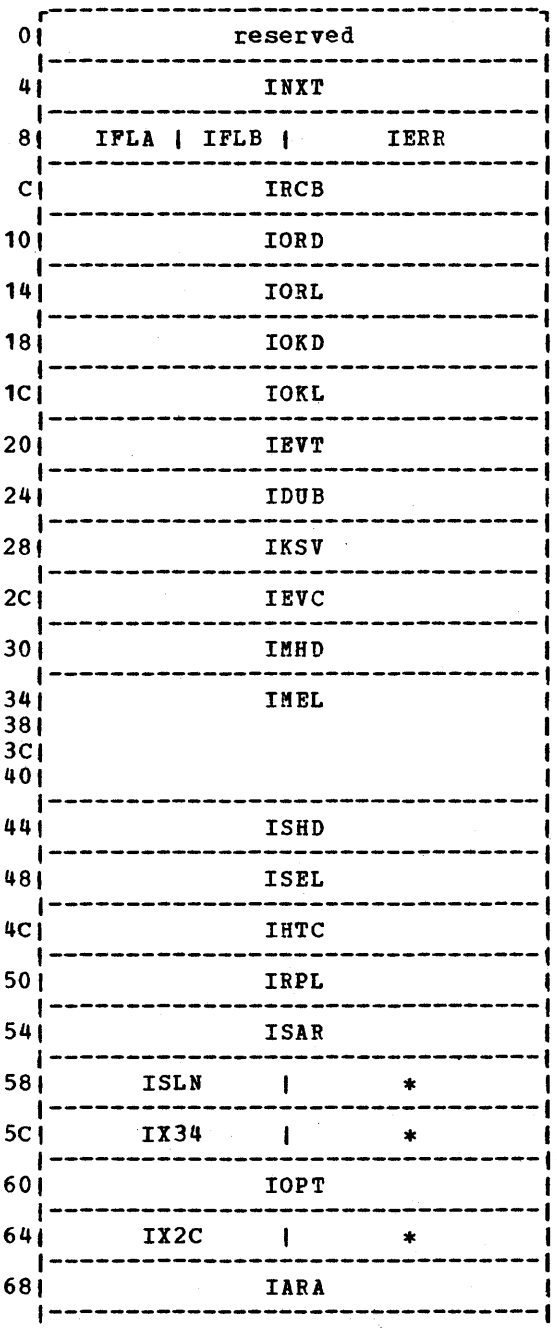
Input Output Control Block (IOCB)

Function
To hold information about the current I/O operation on VSAM files.

When Generated
During the execution of the OPEN statement.

Where Held
In non-LIFO storage.

How Addressed
From the FCB.



* indicates reserved fields

INXT	Next IOCB on chain (set to 0)
IFLA	Flag byte - bits set to '1' indicate: Bits 0 - 3 reserved Bit 4 general error flag Bit 5 unable to complete operation Bits 6 - 7 reserved
IFLB	Code byte containing offset within 'look-up' table used for record checking
IERR	Error codes (as for FER1 & FER2 of FCB see under FCB)
(IER1 IER2)	First byte is for TRANSMIT, second byte for ENDFILE, RECORD, KEY & ERROR conditions
IRCB	Request Control Block
IORD	1st word of record descriptor = record address
IORL	2nd word of record descriptor = flags + record length
IOKD	1st word of key descriptor = key address
IOKL	2nd word of key descriptor = flags + key length
IEVT	A(EVENT variable)
IDUB	A(dummy buffer)
IKSV	A(key save area)
IEVC	1st word of pseudo CCB
MODCB	plist (5 words starting at offset X'30')
IMHD	A(header entry) -> IHTC
IMEL	Element entry addresses (maximum of 4)
SHOWCB	plist (2 words starting at offset X'44')
ISHD	A(header entry)-> IHTC
ISEL	A(element entry)
	Header control entry (4 words

starting at offset X'4C')

IHTC header type code for MODCB/SHOWCB of RPL

IRPL Address of request parameter list

ISAR A(receiving area for SHOWCB)

ISLN Length receiving area for SHOWCB

Element control entries start at offset X'5C' and continue to end of IOCB. Each entry occupies 2 words, with keyword type code set in 1st half-word for example: 1X34 = X'0034'

The 2nd word of each entry is used as either a setting field for MODCB or a receiving field for SHOWCB. The IOCB field names are listed with their corresponding RPL (Request Parameter List) parameters.

IOPT	OPTCD
IARA	AREA
IARL	AREALEN
IRCL	RECLEN
ISIK	FDBK
IARG	ARG
IKYL	KEYLEN

Interlanguage Root Control Block (IBMBILC1)

Function

Connects ZCTL and interlanguage VDA to interlingual routines, and records state of activation of language interfaces.

When Generated

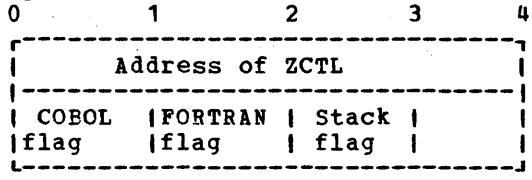
During compilation.

Where Held

In static internal storage, as a control section.

How Addressed

By an offset from register 3 known to compiled code.



COBOL flag = COBOL active

FORTRAN flag = FORTRAN active

Stack flag = PLISA specified

Note: If COBOL or FORTRAN flag is on PL/I is also active.

Interlanguage VDA

Function

To hold information required for interlanguage calls. Used for information that alters from invocation to invocation.

When Generated

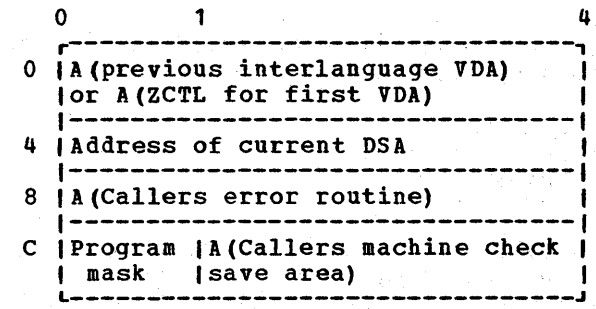
One interlanguage VDA is generated for each interlanguage call made from PL/I to FORTRAN or COBOL. An interlanguage VDA is also acquired if the PL/I environment has not yet been set up when PL/I is called from COBOL or FORTRAN.

Where Held

In the LIFO storage stack.

How Addressed

The latest interlanguage VDA is addressed from offset 0 in ZCTL.



Key Descriptor (KD)

Function

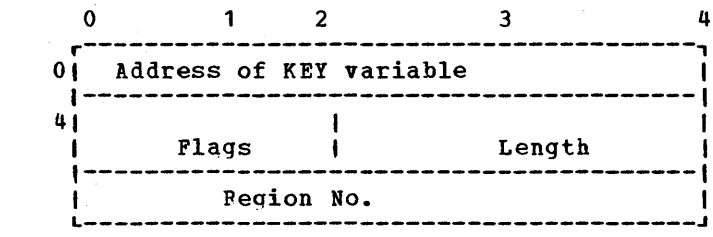
Contains address and length of key for passing to library record I/O routines.

When Generated

As far as possible during compilation. If necessary, completed during execution.

Where Held

Normally in static internal control section. In static external control section if key is EXTERNAL. Will be copied into, or generated in, temporary storage if procedure is reentrant or recursive.



Word 1

a. Address of source key (excluding the length bytes if VARYING)

b. Address of where to put key (excluding length bytes if VARYING)

Word 2

Bit 0	'1'B if KEYTO string is VARYING. (If this bit is set, the I/O transmitters will set the current length field).
Bit 1	'1'B if word 3 contains a region number.
Bits 2-15	Unused (zero)
Bits 16-31	Length of key string (excluding length bytes for VARYING); current length for KEY or KEYFROM, maximum length for KEYTO.

Word 3

Region number in fixed binary, right justified.

Label Data Control Block

Function

Holds the address of the data item and, if a label variable, the address of the associated DSA.

When Generated

Label constants: during compilation
Label variables: when the variable is allocated

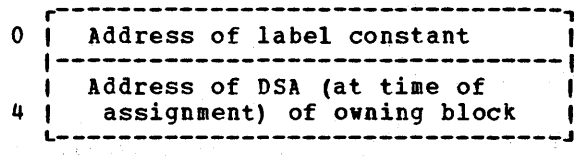
Where Held

Depends on the storage class of the data item

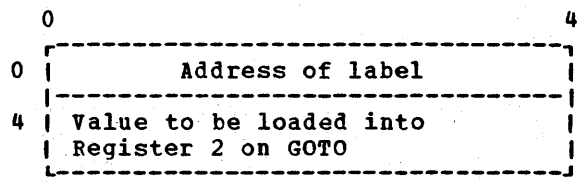
How Addressed

As a variable, dependant on storage class.

Label Variable



Label Constant



Library Work Space (LWS)

Function

Space reserved for two pre-formatted DSAs used by certain library modules.

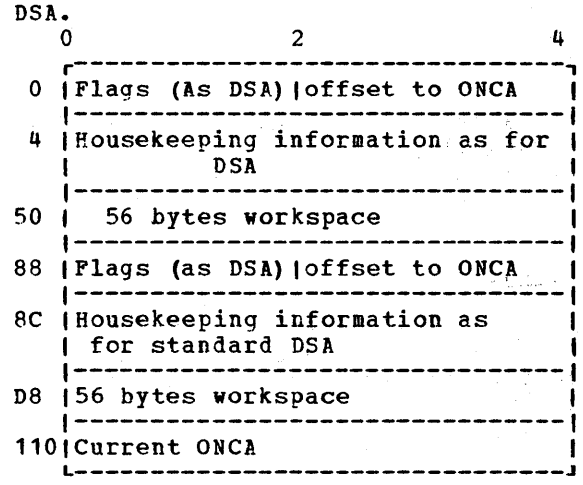
When Generated

The first LWS is generated during program initialization. Subsequent LWSs are allocated before entry to any on-unit. This is because the on-unit may require the use of library modules using LWS but must not alter the environment of the interrupt.

Where Held

First allocation in the program management area. Subsequent allocations in the LIFO storage stack. ONCAs are generated with LWS.

How Addressed The associated allocations is addressed from offset X'48' in the current DSA.



On Communications Area (ONCA)

Function

An area in which built-in function values or their addresses are placed, after the occurrence of a PL/I interrupt.

When Generated

The first ONCA is generated during program initialization. Subsequent ONCAs are generated with each allocation of LWS.

Where Held

Contiguous with LWS in the program management area and in the LIFO stack.

How Addressed By an offset held from the start of LWS held at offset X'02' in each segement of LWS.

Dummy ONCA

The dummy ONCA has the same format as other ONCAs and holds default values for those condition built-in functions that have default values.

Flags1

Bit 0 = 0 ONFILE invalid
= 1 ONFILE valid

Bit 1 = 0 ONCHAR/ONSOURCE invalid
= 1 ONCHAR/ONSOURCE valid

Bit 2 = 0 ONIDENT invalid
= 1 ONIDENT valid

Bit 3 = 0 ONKEY invalid
= 1 ONKEY valid

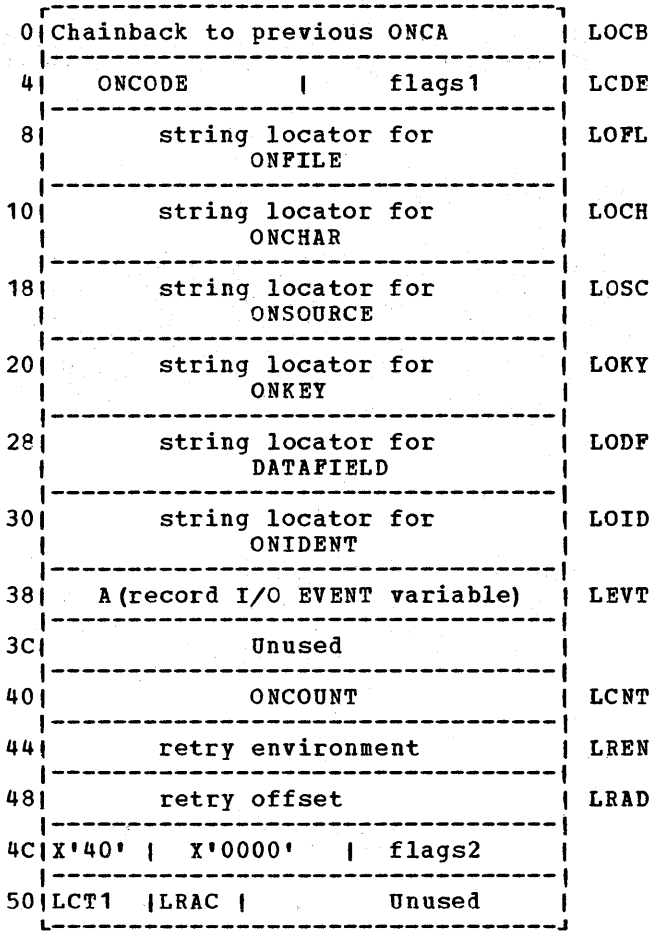
Bit 4 = 0 DATAFIELD invalid
= 1 DATAFIELD valid

Bit 5 = 0 No associated EVENT variable
= 1 Associated EVENT variable

Bit 6 Unused

Bit 7 = 0 ONCOUNT invalid
= 1 ONCOUNT valid

Bits 8-15 unused



Flags 2

Bit 0 = 0 ONSOURCE/ONCHAR not used in on-unit
= 1 ONSOURCE/ONCHAR used

Bit 1 = 0 ONSOURCE not set in ONCA
= 1 ONSOURCE set in ONCA

Bits 2-7 unused

LCT1

Copy of TCA flag byte 1 (TFB1)

LRAC

Retry address code

Retry offset

The offset from the base of the library module involved to the address at which a conversion will be reattempted if ONSOURCE or ONCHAR has been used.

On Control Block (ONCB)

Function

Contains pointer to associated on unit, or indicates action to be taken when interrupt occurs.

When Generated

Static ONCBs are generated during compilation, one for each ON statement. Dynamic ONCBs are generated by the prologue code of the procedure or block in which the ON statement occurs, or are allocated in a VDA when the ON statement is executed.

Where Stored

Static ONCBs are generated in the static internal control section. Dynamic ONCBs are stored in the DSA of the block in which the associated on-unit occurs.

How Addressed

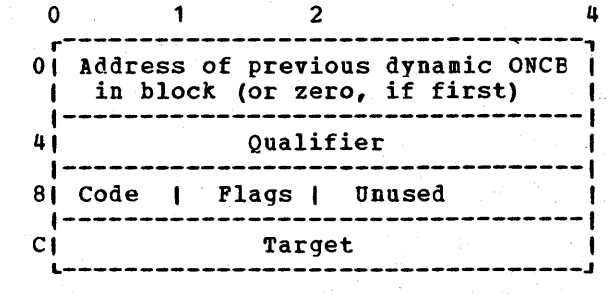
Start of dynamic ONCBs - offset X'60' in the DSA.

First Static ONCB - offset X'5C' in DSA.

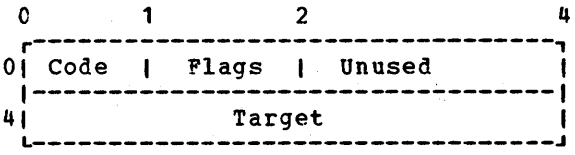
Static and Dynamic ONCBs

Static ONCBs are generated for unqualified conditions. Dynamic ONCBs are generated for qualified conditions (ENDPAGE, ENDFILE, etc.)

Dynamic ONCB



Static ONCB



Qualifier

A(FCB) for I/O conditions
A(SYMTAB) for CHECK
A(CSECT) for CONDITION condition.

Code

PL/I code for condition

Flags

Bit 0 = 0 SYSTEM not specified
1 SYSTEM specified

Bit 1 = 0 Not a null on-unit
1 Null on-unit

Bit 2 = 0 Not a GOTO only on-unit
1 GOTO only on-unit

Bit 3 = 0 Condition not established
1 Condition established

Bit 4 Unused

Bit 5 = 0 Condition not enabled at block entry
1 Enabled at block entry

Bit 6 = 0 Condition disabled
1 Condition enabled

Bit 7 = 0 SNAP not specified
1 SNAP specified

Target

Address of on-unit, or offset in DSA of word containing A(label variable or label temporary).

Open Control Block

Function

Used to indicate that a file attribute (either input or output) was declared in the associated OPEN statement.

When Generated

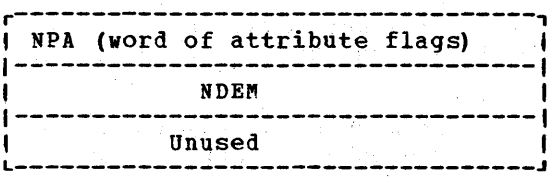
During compilation.

Where Held

Static internal control section.

How Addressed

By an offset from register 3 known to compiled code.



NPA - Open attributes

This word indicates the explicit and implied attributes on the OPEN statement.

Byte No.	Hex. Value	Attributes
1	Not used	
2	10	INPUT
	20	OUTPUT
3,4	Not used	

NDEM - Open conflict mask

This is a mask generated by the compiler containing bits for all attributes which conflict with those on the OPEN statement.

PLIMAIN

Function

To hold address of entry point of a PL/I main procedure.

When Generated

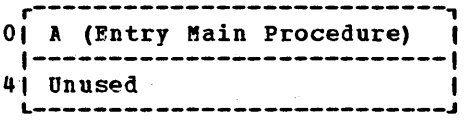
During compilation of a procedure with the MAIN option.

Where held

Separate control section.

How Addressed

As a control section.



PLISTART

Function

Entry point for PL/I program, passing control to IBMDPIR. Primary entry point passes control to IBMDPIRA. PLICALLA passes control to IBMDPIRB. PLICALLB passes control to PLICALLC.

When Generated

During compilation for every PL/I compilation.

Where held

Held as a separate control section.

How Addressed

As a control section.

```
PLISTART CSECT
         EXTRN  PLIMAIN
         BALR   15,0
         USING  *,15
         L      15,PIR
         BALR   0,15
         DC     A PLIMAIN
PIR      EQU    *
         DC     V IBMBPIRA
         END    PLISTART
```

Record Descriptor (RD)

Function

Contains address and length of record for passing to library record I/O routines.

When Generated

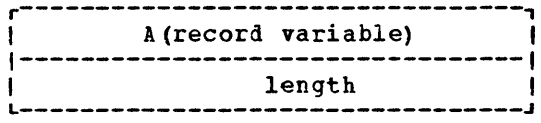
As far as possible during compilation. If necessary, completed during execution.

Where Held

Normally in static internal control section. In static external control section if record is EXTERNAL. Will be copied into, or generated in, temporary storage if procedure is reentrant or recursive.

How Addressed

By an offset from register 3 known to compiled code.



Word 1

1. Address of the data to be written out.

2. Address of where data read in is to be put.

3. LOCATE statement: Address of where to store buffer address.

   READ SET statement: Address of pointer to be set.

READ IGNORE statement: Ignore factor.

Word 2

Bits 0 - 7 indicate the type of INTO or FROM argument as follows:

- X '00' fixed length strings
- X '01' area variables
- X '02' varying length character strings
- X '03' varying length bit strings

Bits 8-31 length of data to be transmitted (length of variable or buffer for locate mode).

The value is in bytes for all strings including bit strings.

For VARYING strings, the value includes the two length bytes, and is the maximum length for input operations and for LOCATE, the current length for other operations.

Request Control Block (RCB)

Function

Used by the record I/O interface module (IBMDRIO) to check the validity of an I/O statement. The instruction in RTMI is carried out by IBMDRIO.

When Generated

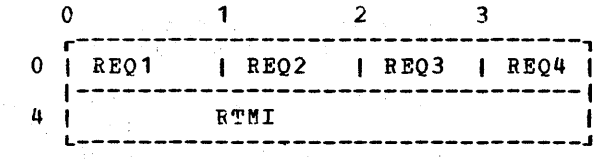
During compilation.

Where Held

Static internal control section.

How Addressed

From parameter list passed in register 1 to IBMDRIO.



REQ1 (statement identification)

- 00 - READ
- 04 - REWRITE
- 08 - WRITE
- 0C - LOCATE
- 10 - DELETE
- 14 - UNLOCK
- 18 - WAIT

REQ2 (options)

- 80 - INTO/FROM
- 40 - SET
- 20 - IGNORE
- 02 - NOLOCK
- 01 - EVENT

REQ3 (options)

- 01 - KEY
- 02 - KEYTO
- 04 - KEYFROM

REQ4 unused

RTMI

Either a TM or a BR instruction depending on source program.

A TM instruction is used if the statement cannot be checked for validity during compilation, or if it has been checked and found to be invalid.

The TM instruction is used by IBMDRIO for testing the validity of a statement and is;

X'91MM2SSS'

where MM is byte containing current statement bit and SSS is offset of corresponding byte in FCB statement mask.

A BR instruction is used if the statement has been checked during compilation and found to be valid.

Unconditional branch instruction to PL/I library or LIOCS transmitter.

Statement Frequency Count Table

Function

To retain a record of the number of times a statement has been branched to or from, for use by the COUNT option.

When Generated

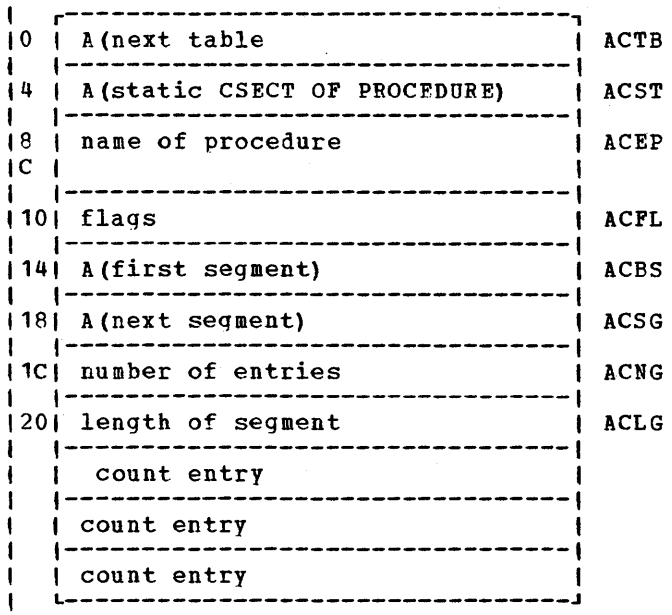
When the associated external procedure is entered.

Where Held

Non-LIFO storage.

How Addressed

The statement frequency count table for the first external procedure in a program is addressed from offset X'80' in the TCA appendage (TIA). The tables are chained together and the chain field of the last table set to zero. The chain field is at offset 0 in the table. The most recently used table is addressed from X'84' in the TIA.



ACBS The address held in ACBS is the address of ACSG. If tables are segmented, second and subsequent sections of the table will start at a point equivalent to ACSG.

ACFL Flags

ACBI	Bit 0	last update was for a branch in
ACGT	Bit 1	last update was for a GOTO out of block
ACIA	Bit 2	table inactive
ACNM	Bit 3	not used
ACUI	Bit 4	not used
ACZC	Bit 5	not used

Other bits unused.

Stream I/O Control Block (SIOCB)

Function

Holds addresses of source and target, source and target DEDs etc and is used as parameter list by stream I/O routines.

When Generated

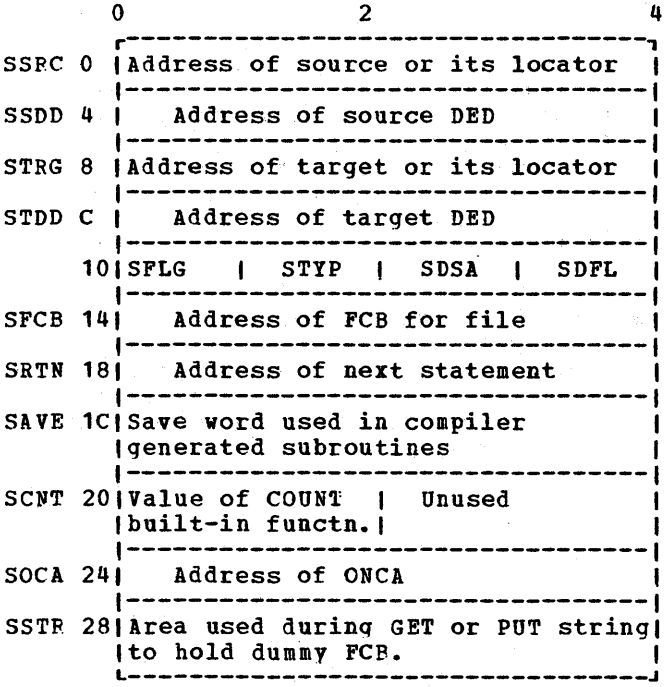
During execution for the duration of the stream I/O statement.

Where Held

In temporary storage.

How Addressed

By register 1 during the stream I/O statement.



Flag Byte SFLG

Bit 0 = 1 Transmit on input

Bit 1 = 1 VDA used in edit-directed input

Bit 2 = 1 IBMDSED is used

Bit 3 = 1 Call to IBMBSIST required after dealing with next item (GET or PUT STRING only)

SDSA

DSA level number (used only for data-directed I/O)

Type code STYP

Bit 0 = 1 data-directed I/O

Bit 1 = 1 list-directed I/O

Bit 2 = 1 edit-directed I/O

Bit 3 = 1 string I/O

Bit 4 = 1 CHECK entry to data-directed I/O

Bit 5 = 1 input

Data-directed flag SDFL

Bit 0 = 1 Terminating call to data-directed output

Statement Number Table (DST)

Function

To relate statement numbers to offsets so that statement numbers may be given in execution-time messages.

When Generated

During compilation, if the GOSTMT option is in effect.

Where Held

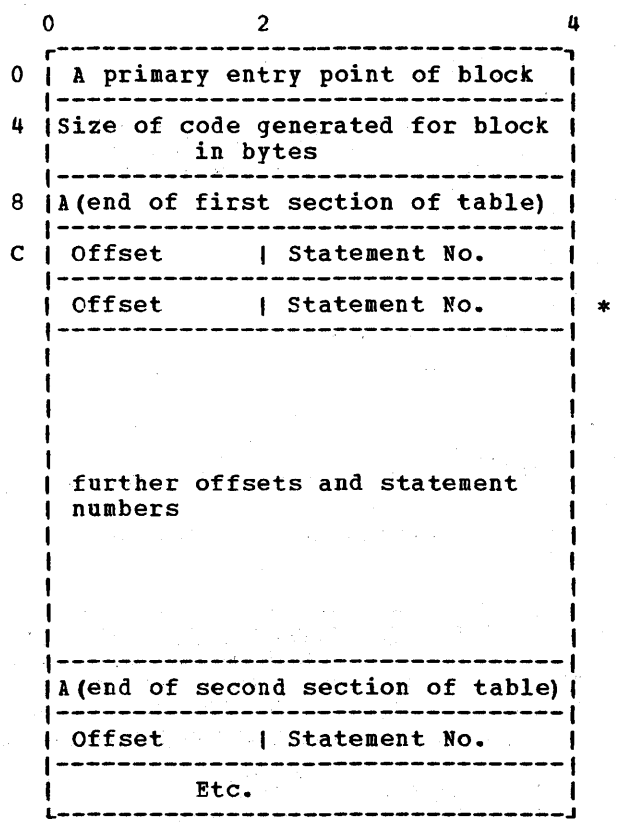
Static internal control section.

How Addressed

From offset 8 from each principal entry point to a block.

Sections of Table

As offsets are held in two bytes and the value may in fact take up to three bytes (4096), it is necessary to hold the table in sections. If the offset is greater than X'7FFF' the statement number will be held in the second or subsequent sections of the table. Obtain the number given by translating the offset into binary and ignoring the last 15 bits and step down this number of sections of the table. (For example, if the offset was X'8FFF', translate to binary = '1000 1111 1111 1111'B, ignore last 15 binary digits =1, therefore step down one section of the table. If the offset was X'18FFF' the binary would be '0001 1000 1111 1111 1111'B. Ignoring the 15 right-hand bits leaves '11'B therefore step down three sections of the table.)



* = End of first section

Offset: Offset is the offset of the first byte of the statement relative to the address of the primary entry point of the block.

String Locator/descriptor

Function

Used to pass the address and the length of strings to other routines. Also for handling strings with adjustable lengths (e.g., DCL STRING CHAR (N)).

When Generated

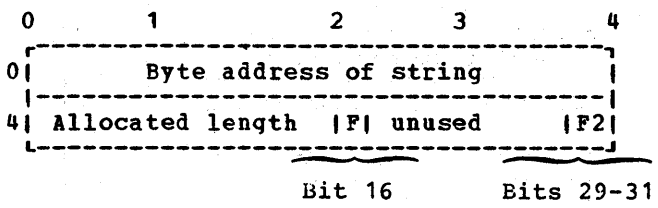
Storage reserved during compilation. Fields completed during execution if string has adjustable length.

Where Held

Static internal control section.

How Addressed

By an offset from register 3 known to compiled code.



Word 2
Bit 16

F = '0' B Fixed string (First bit of second byte)
'1' B Varying string

Bits 29-31

F2 Used for bit strings to hold offset from byte address of first bit in string (3 bits)

Allocated length

For varying strings this is the declared length. Length is held in bits for bit strings and in bytes for character strings.

STRING DESCRIPTOR

The string descriptor is the second word of the string locator/descriptor. It appears in structure descriptors and in the description field of controlled variables.

Structure Descriptor

Function

Contains information about the offset of each element within a structure, and the nature of each element. Used when passing a structure to another routine, or for accessing structure elements during execution, if the structure is declared with adjustable extents or with the REFER option.

When Generated

If the structure has no adjustable elements, during compilation. If the structure has adjustable elements, during execution from information held in the aggregate descriptor descriptor.

Where Stored

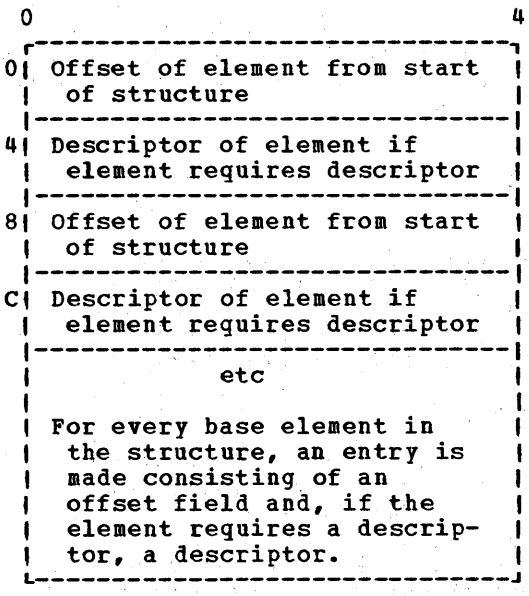
Static internal control section.

How Addressed

By an offset from register 3 known to compiled code.

General Format

For each base element in the structure, a fullword field containing the offset of the start of the element from the start of the structure is given. If the base element is a string, area, or array, this fullword is followed by the offset field for the next base element.



Offset

The offset field is held in bytes, Any adjustments needed for bit-aligned addresses are held in the respective descriptors.

Symbol Table (SYMTAB)

Function

Holds the name of the variable during execution and associates it with the address of the variable. Used only when data-directed I/O or the CHECK condition is specified.

When Generated

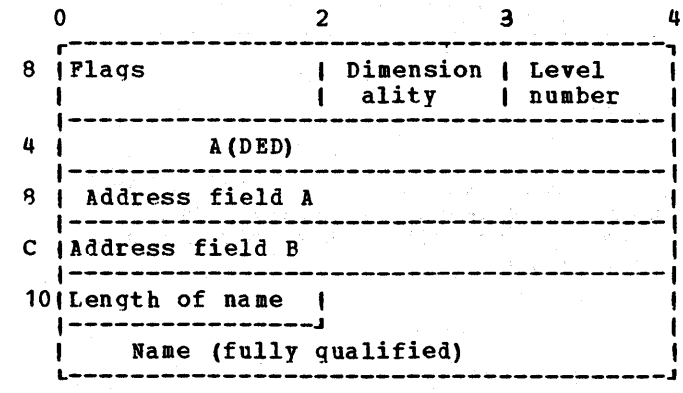
During compilation, if data-directed I/O or the CHECK condition is used in the program

Where Held

Static internal control section for internal names. Separate control section for external names. For external variables the name of the control section is the name of the variable followed by an *.

How Addressed

By an offset from register 3 known to compiled code for internal variables. As a control section for internal variables.



Flags

Bits 0,1 & 2 = '000'B STATIC
= '100'B AUTOMATIC
= '010'B CONTROLLED (not param.)
= '001'B BASED
= '011'B DEFINED
= '101'B a non-CONTROLLED parameter
= '111'B a CONTROLLED parameter

Bit 3 = '1'B EXTERNAL
= '0'B INTERNAL

Bit 4 = '1'B item may appear in some CHECK list.
= '0'B item appears in no CHECK list.

(Bit 4 must be '1'B if item is EXTERNAL).

Bit 5 = '1'B Address field A refers to data.
= '0'B Address field A refers to locator.

(Bit 5 must be '0'B for a CONTROLLED parameter)

Bit 6 = '1'B a member of a structure.
= '0'B not a member of a structure.

Bit 7 = '1'B Normal SYMTAB.
= '0'B Short SYMTAB (has fields A & B omitted).

Bit 8 = '1'B Address field A addresses code.
= '0'B Address field A does not address code.

Bits 10 - 11 reserved: must be set to '0'B.

Bit 12 = '1'B Symtab concerns a BASED variable; Bits 0,1,2,5,8 of Flags, level # and Field A all refer to the POINTER qualifier.
= '0'B normal Symtab.

Bit 13 = '1'B Symtab concerns a BASED variable and Field B contains an address (in Static).
= '0'B If Symtab concerns a BASED variable, Field B contains an offset (right justified) in the DSA defined by level #.

Bits 14,15 reserved: must be set to '0'B.

Dimensionality

The number of dimensions declared for an array item. Dimensionality is zero for other items.

Level number

(for AUTOMATIC, DEFINED, and BASED items. Also for all parameters.) The level of the block in which the variable is declared. The level of a block is one greater than the level of the immediately containing block; the level of the external block is 0.

Address Fields

Addresses are held in different formats for different data types. As far as possible, addresses are held in address field A. However, more information than can be held in a fullword field is sometimes required. When this is the case, address field B is also used.

Address field A

If STATIC — Address of data or address of locator for items that have locators.

If AUTOMATIC — Offset within the associated DSA of the data or of the locator for items that have locators

If CONTROLLED — Address of anchor word.

If BASED — Offset of one word field with in associated DSA containing address of declared pointer qualifier.

If PARAMETER or DEFINED — Offset of one word field in associated DSA containing address of corresponding argument, or DEFINED data, or its locator. For CONTROLLED parameters, the argument is its anchor word.

Address field B

If non-structured AUTOMATIC, STATIC, DEFINED or CONTROLLED parameter, field B is set to a fullword of zeros.

If structured not BASED — Offset from start of structure descriptor to field that holds offset of element from start of structure. See "Structure Descriptor."

If BASED (except when flag bit 12 or 13 is set) — For non-structured BASED items field B holds the offset of the descriptor from the start of the DSA in which it is held.

For structured BASED items, the offset is to the offset word in the structure descriptor. This word holds the offset of the item from the start of the structure. See "Structure Descriptor".

Length

Length is the number of characters in the fully qualified name.

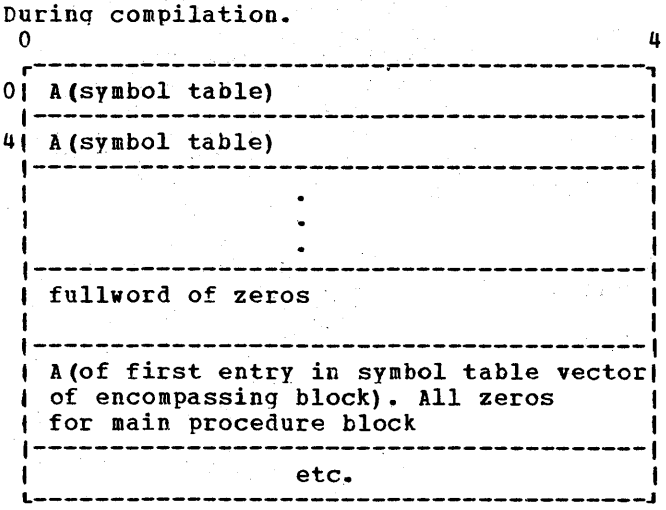
Symbol Table Vector

Function

Holds addresses of symbol tables and associates them with the block in which the associated names were declared.

When Generated

During compilation.



General Format

The format of symbol table vector is a series of fullwords. These contain either:

1. The address of a symbol table

or

2. The address in the symbol table vector of the start of the entries for the encompassing block.

or

3. A fullword of zeros indicating the end of the current block.

Where Held

Static internal control section.

How Addressed

By an offset from register three known to compiled code.

Task Communications Area (TCA)

When Generated

During program initialization by IBMDPIR.

Function

Acts as a central communications area for the program. Contains addresses of essential routines and control blocks, and various flags. (See chapter 5).

How Addressed

By register 12 -43

Where Held

In the program management area at the head of the initial storage area (ISA).

0	Flags	TFLG
4	Unused	
8	Segment # | BOS	TBOS
C	Segment # | EOS	TEOS
10	Unused	
14	A(current event variable)	TEVT
18	A(External Save Area)	TESA
1C	A(TRT Table)	TTRT
20	A(overflow routine - get DSA)	TOVF
24	Branch to get DSA subroutine	TGDS
28	A(TCA appendage)	TTIA
2C	A(error handler)	TERR
30	A(Save Area for Overflow Routine)	TPSA
34	Open File Chain Anchor	TFOP
38	Branch instruction for TCA GOTO code	TGTC
3C	A(Bugtable)	TBUG
40	A(Diagnostic File Block)	TDFB
TURC 44	PL/I Return Code | User return Code	TORC
48	A(Overflow Routine for Get VDA)	TOVV
4C	A(Flow stmt number table)	TFST
50	A(Tab table)	TTAB
54	A(Flow module)	TEFL
58	Branch instruction for call routine Branch instruction for link routine	TCAL TLIN
5C	Unused	
60	Unused | TRLR | TTLR	TENV
64	(Set to zero)	TPRI

68	Unused	
6C	A(Get Dynamic Storage Routine)	TGET
70	A(Free Dynamic Storage Routine)	TFRE
74	A(Overflow Routine for Get DSA)	TOVFO
78	A(Error Handler)	TERRO
7C	Environment Description	TENVO
80	Normal GOTO Code Used when GOTO out of block may occur	TGTCO
F0	A(IBMBEFLC)/ dummy if NOFLOW and NOCOUNT	TEFC
F4	A(Interpretive GOTO routine)	TGTM
F8	Unused	
FC	Unused	
100	Unused	
104	Unused	
108	A(WAIT routine)	TAWT
10C	A(COMPLETION pseudovariable routine)	TACP
110	A(EVENT assign routine)	TAEA
114	Unused	
118	2 unused words	

TFLG contains flag bytes TFB0, TFB1, TFB2, and TFB3.

	Bit	
TFB0	all	not used in this implementation
TFB1	0	not used
	1='1'B	Event I/O on-unit active
	2	not used
	3='1'B	Abnormal exit requiring special action
	4-7	not used
TFB2	0='1'B	Raise SIZE for fixed-point divide, fixed-point overflow, exponent overflow, decimal overflow exceptions
	1='1'B	Ignore the exceptions detailed for bit 0
	2-6	not used
	7='1'B	I/O conversion
TFB3	all	not used

TENV contains environment description
TRLR resident library release number
TTLR transient library release number

TOVFO TERRO TENVO TGTCO

These fields are used in previous releases and are retained for compatibility.

TCA Appendage (TIA)

Function

To hold control and communication information.

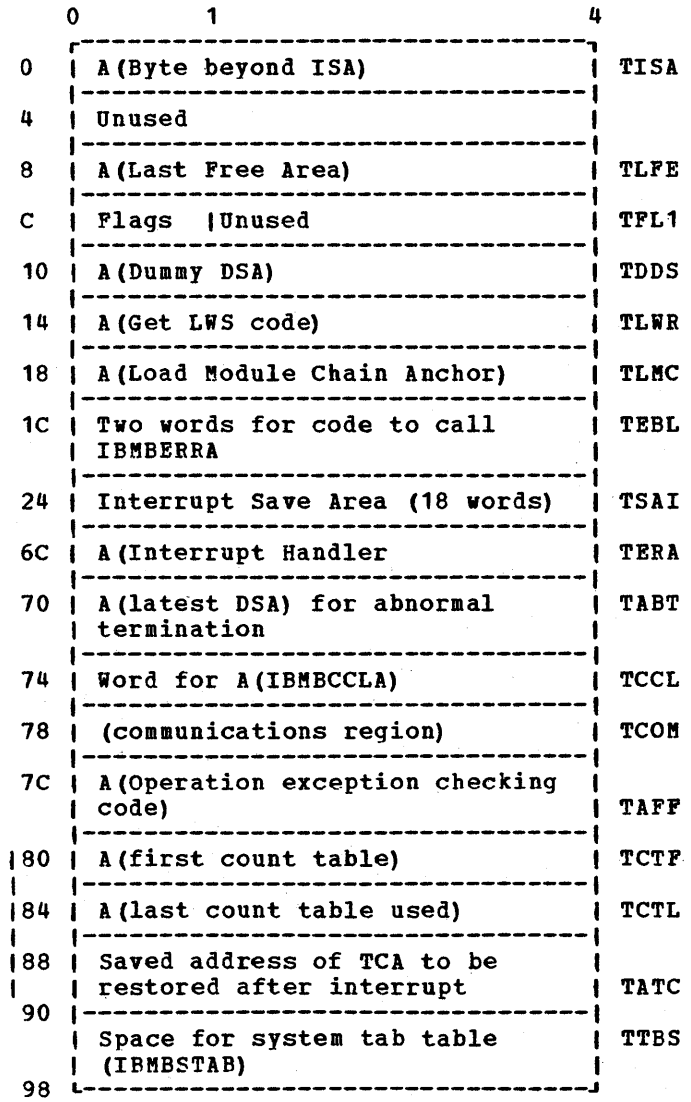
When Generated

During program initialization.

Where Held

Program management area.

How Addressed
From Offset X'28' in the TCA



Flags - TFL1

TFLS Bit 1 = 0 SYSPRINT not open for STREAM PRINT
= 1 SYSPRINT open STREAM PRINT

TFLJ Bit 2 = 0 Abnormal termination exit not in progress
= 1 Abnormal termination exit in progress

TFLK Bit 3 = 0 No dump I/O in progress
= 1 Dump I/O in progress

Notes on various fields

TISA identifies end of region.

Note: Chain beginning in TLFE is continued at offset 4 in free area. First word in free area contains length.

TLWR is an address in IBMDPIR that acquires library workspace.

TFRA is entry point A of IBMDERR.

TCCL is a field used to hold the address of the complex string conversion routine IBMBCCL. This routine is in the transient library and is loaded by the bootstrap routine IBMBCCS. When the routine is loaded, the address is placed in the field TCCL.

TAFF is the address of the code used to check whether an operation interrupt is caused by an attempt to execute a floating point instruction on a machine with no floating point hardware.

Zygo-Lingual Control List (ZCTL)

Function

To hold information required for interlanguage calls. Holds information that does not change for every invocation.

When Generated

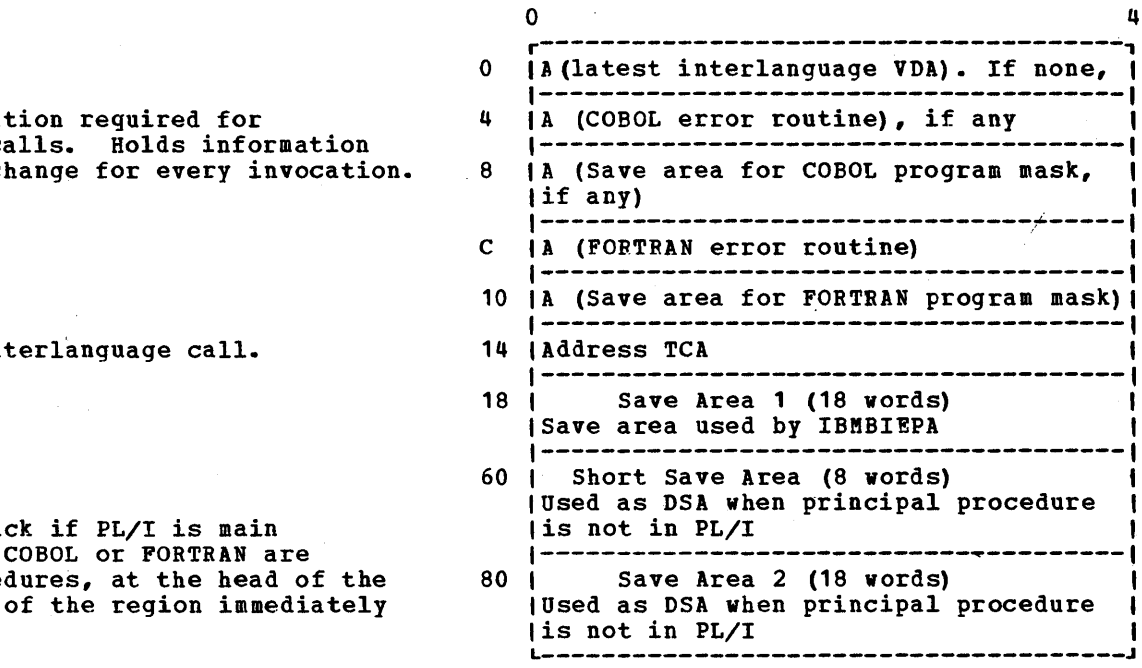
On the first interlanguage call.

Where Held

In the LIFO stack if PL/I is main procedure. If COBOL or FORTRAN are principal procedures, at the head of the unused portion of the region immediately before the TCA.

How Addresses

From offset X'0' in IBMBILC1 the Interlangauge Root Control Block.



Appendix C: List of PL/I Library Modules

The following list of modules is arranged in alphabetical order of the last three letters of the module name. This ordering is used to save the reader the trouble of remembering whether the module is prefixed with IBMB or IBMD.

Resident Library Modules

Name	Function	Size (approx)
IBMBAAH	ALL, ANY (simple and interleaved arrays)	390 bytes
IBMBAIH	Indexer for interleaved arrays	100 bytes
IBMBAMM	Structure mapping	1760 bytes
IBMBANM	STRING built-in function	1630 bytes
IBMBAPC	PROD (arrays with fixed point integer elements)	580 bytes
IBMBAPF	PROD (arrays with floating point elements)	370 bytes
IBMBAPM	STRING pseudovariable	1230 bytes
IBMBASC	SUM (arrays with fixed-point elements)	420 bytes
IBMBASF	SUM (arrays with floating-point elements)	330 bytes
IBMBAYF	POLY built-in function	380 bytes
IBMBBBA	AND, OR operations (byte-aligned bit strings)	520 bytes
IBMBBBN	NOT operation (byte-aligned bit strings)	400 bytes
IBMBBCI	INDEX (character strings)	200 bytes
IBMBBCK	Concatenate, REPEAT (character strings)	610 bytes
IBMBBCT	TRANSLATE (character strings)	770 bytes
IBMBBCV	VERIFY (character strings)	210 bytes
IBMBBGB	BOOL (bit strings)	660 bytes
IBMBBGC	Compare (general bit strings)	240 bytes
IBMBBGF	Assign (byte-aligned bit strings) and Fill (general bit strings)	390 bytes
IBMBBGI	INDEX (bit strings)	350 bytes
IBMBBGK	Concatenate, REPEAT, General Assign (bit strings)	890 bytes
IBMBBGS	SUBSTR SLD	330 bytes
IBMBBGV	VERIFY (bit strings)	420 bytes
IBMBCAC	Conversion director (arithmetic to character)	700 bytes
IBMBCBB	Conversion (bit to bit)	350 bytes
IBMBCBC	Conversion (bit to character)	220 bytes
IBMBCBQ	Conversion (bit to pictured character)	240 bytes
IBMBCCA	Conversion director (character to arithmetic)	520 bytes
IBMBCCB	Conversion (character to bit)	420 bytes
IBMBCCC	HIGH, LOW, Assign (character strings)	270 bytes
IBMBCCQ	Conversion (character to pictured character)	410 bytes
IBMDCCS	String conversion director bootstrap	340 bytes
IBMBCE	Conversion (fixed decimal - free decimal - float - fixed binary)	720 bytes
IBMBCGP	Check input (pictured decimal)	870 bytes
IBMBCGQ	Check input (pictured character)	200 bytes
IBMBCGT	Table of powers of ten	140 bytes
IBMBCGZ	Set a subfield of a complex number to zero	300 bytes
IBMBCH	Conversion (fixed binary - float - free decimal)	480 bytes
IBMBCK	Conversion (fixed decimal - free decimal - fixed decimal)	370 bytes
IBMBCM	Conversion (pictured decimal to packed decimal)	810 bytes
IBMBCO	Conversion (packed decimal to pictured decimal)	1080 bytes
IBMBCP	Conversion (bit to fixed binary or float)	490 bytes
IBMBCR	Conversion (fixed binary or float to bit)	400 bytes
IBMBCT	Conversion (decimal constant to packed decimal)	670 bytes
IBMBCU	Conversion (binary constant to float)	780 bytes
IBMBCV	Conversion (packed decimal to E format)	670 bytes
IBMBCW	Conversion (packed decimal to F format)	490 bytes
IBMBCY	Conversion (fixed binary to fixed binary and float to float)	250 bytes
IBMDEFL	FLOW and COUNT option	1200 bytes
IBMBEOC	ON-code	220 bytes
IBMBEOL	ONLOC built-in function	160 bytes
IBMBERC	CHECK system action	490 bytes

IBMDERR	Error handler	1500 bytes
IBMDEVO	Event Variable operations	16 bytes
IBMDIEC	Interlanguage housekeeping	500 bytes
IBMDIEF	Interlanguage housekeeping	910 bytes
IBMDIEP	Interlanguage housekeeping	1000 bytes
IBMDJDS	DISPLAY	740 bytes
IBMDJDT	DATE built-in function	80 bytes
IBMDJDY	DELAY	130 bytes
IBMDJD7	DISPLAY without EVENT	590 bytes
IBMDJTT	TIME built-in function	150 bytes
IBMDJWI	WAIT (array events)	390 bytes
IBMDJWT	WAIT (multiple events)	800 bytes
IBMGJWT	WAIT (single event)	220 bytes
IBMDKCP	Checkpoint/restart interface	820 bytes
IBMDKDM	Dump bootstrap	140 bytes
IBMDKST	SORT interface	1440 bytes
IBMBMAL	SQRT (long float real)	170 bytes
IBMBMAS	SQRT (short float real)	170 bytes
IBMBMAX	SQRT (short float complex)	290 bytes
IBMBMAY	SQRT (long float complex)	300 bytes
IBMBMBL	EXP (long float real)	460 bytes
IBMBMBS	EXP (short float real)	260 bytes
IBMBMBX	EXP (short float complex)	140 bytes
IBMBMBY	EXP (long float complex)	140 bytes
IBMBMCL	ERF, ERFC (long float real)	640 bytes
IBMBMCS	ERF, ERFC (short float real)	410 bytes
IBMBMDL	LOG, LOG2, LOG10 (long float real)	340 bytes
IBMBMDS	LOG, LOG2, LOG10 (short float real)	260 bytes
IBMBMDX	LOG (short float complex)	230 bytes
IBMBMDY	LOG (long float complex)	230 bytes
IBMBMGL	SIN, SIND, COS, COSD (long float real)	390 bytes
IBMBMGS	SIN, SIND, COS, COSD (short float real)	310 bytes
IBMBMGX	SIN, SINH, COS, COSH (short float complex)	310 bytes
IBMBMGY	SIN, SINH, COS, COSH (long float complex)	370 bytes
IBMBMHL	TAN, TAND (long float real)	320 bytes
IBMBMHS	TAN, TAND (short float real)	260 bytes
IBMBMHX	TAN, TANH (short float complex)	230 bytes
IBMBMHY	TAN, TANH (long float complex)	230 bytes
IBMBMIL	SINH, COSH (long float real)	240 bytes
IBMBMIS	SINH, COSH (short float real)	160 bytes
IBMBMJL	TANH (long float real)	260 bytes
IBMBMJS	TANH (short float real)	200 bytes
IBMBMKL	ATAN, ATAND (long float real)	470 bytes
IBMBMKS	ATAN, ATAND (short float real)	360 bytes
IBMBMKX	ATAN, ATANH (short float complex)	260 bytes
IBMBMKY	ATAN, ATANH (long float complex)	260 bytes
IBMBMLL	ATANH (long float real)	260 bytes
IBMBMLS	ATANH (short float real)	180 bytes
IBMBMML	ASIN, ACOS (long float real)	350 bytes
IBMBMMS	ASIN, ACOS (short float real)	260 bytes
IBMBMOD	ADD (fixed decimal real or complex)	290 bytes
IBMBMPU	MULTIPLY (fixed binary complex)	290 bytes
IBMBMPV	MULTIPLY (fixed decimal complex)	280 bytes
IBMBMQU	DIVIDE (fixed binary complex)	460 bytes
IBMBMQV	DIVIDE (fixed decimal complex)	580 bytes
IBMBMRU	ABS (fixed binary complex)	210 bytes
IBMBMRV	ABS (fixed decimal complex)	540 bytes
IBMBMRX	ABS (short float complex)	120 bytes
IBMBMRY	ABS (long float complex)	130 bytes
IBMBMUD	Shift and assign/load (fixed decimal real)	360 bytes
IBMBMVU	Multiplication and Division (fixed binary complex)	290 bytes
IBMBMVV	Multiplication and Division (fixed decimal complex)	660 bytes
IBMBMVW	Multiplication (long and short float complex)	120 bytes
IBMBMWX	Division (short float complex)	100 bytes
IBMBMWY	Division (long float complex)	100 bytes
IBMBMXL	Integer exponentiation (long float real)	140 bytes
IBMBMXS	Integer exponentiation (short float real)	140 bytes
IBMBMXW	Integer exponentiation (short and long float complex)	410 bytes

IBMBMYL	General exponentiation (long float real)	160 bytes
IBMBMYS	General exponentiation (short float real)	150 bytes
IBMBMYX	General exponentiation (short float complex)	260 bytes
IBMBMYY	General exponentiation (long float complex)	270 bytes
IBMDOCL	OPEN/CLOSE bootstrap	240 bytes
IBMDOCN	OPEN/CLOSE address list and record I/O error bootstrap	200 bytes
IBMBPAF	Controlled variable management	150 bytes
IBMBPAM	AREA management	540 bytes
IBMBPGO	Reset CHECK enablement	40 bytes
IBMDPGR	Storage management	610 bytes
IBMDPIR	Program initialization from system	420 bytes
IBMDPJR	Program initialization from caller	330 bytes
IBMDPOV	Overlay	110 bytes
IBMBPRC	Return code module.	40 bytes
IBMDRIO	Record I/O interface module	80 bytes
IBMBSAI	Input conversion director (A, P, and B formats)	420 bytes
IBMBSAO	Output conversion director (A format)	130 bytes
IBMBSBO	Output conversion director (character-P and B formats)	400 bytes
IBMBSCI	Input conversion director (C format)	300 bytes
IBMBSCO	Output conversion director (C format)	290 bytes
IBMDSCP	COPY	230 bytes
IBMDSCV	Conversion fix-up bootstrap	90 bytes
IBMDSDI	Data-directed input	2090 bytes
IBMDSDJ	Data-directed input	2090 bytes
IBMDSDO	Data-directed output	1210 bytes
IBMDSED	Edit-directed I/O housekeeping	1050 bytes
IBMDSEE	Edit-directed combination module	1420 bytes
IBMDSEH	Edit-directed combination subset module	880 bytes
IBMDSEI	Edit-directed input	440 bytes
IBMDSEO	Edit-directed output	210 bytes
IBMBSFI	Input conversion director (F and E formats)	240 bytes
IBMBSFO	Output conversion director (F and E formats)	210 bytes
IBMDSII	GET FILE initialization	420 bytes
IBMDSIO	PUT FILE initialization	330 bytes
IBMDSIS	GET or PUT STRING initialization	350 bytes
IBMDSLI	List-directed input	2220 bytes
IBMDSLJ	List-directed input	2070 bytes
IBMDSLO	List-directed output	1610 bytes
IBMBSMW	Missing output width module	340 bytes
IBMBSPI	Input conversion director (P format)	370 bytes
IBMDSPL	PAGE, LINE, and SKIP	530 bytes
IBMBSPO	Output conversion director (P format)	290 bytes
IBMDSTF	Stream input transmitter	440 bytes
IBMDSTI	Stream print F-format transmitter	190 bytes
IBMDSXC	X and COLUMN format items	440 bytes
IBMBTOC	COMPLETION pseudovariable and Event variable assignment	130 bytes

Transient Library Modules

The following list is arranged in alphabetical order of the last three letters of the module name. This ordering is used to save the reader the trouble of remembering whether the module is prefixed with IBMB or IBMD.

Name	Function	Size (approx)
IBMBCCL	Conversion director (complex strings)	1830 bytes
IBMBCCR	Conversion director (ncn_complex strings)	940 bytes
IBMBEOC	On-code translate	240 bytes
IBMBETA	Miscellaneous non-ON messages (1)	710 bytes
IBMBETB	Miscellaneous non-ON messages (2)	1140 bytes
IBMBETC	Misc. and computational non-ON messages	1000 bytes

IBMBETI	I/O non-ON messages	1340 bytes
IBMBETO	ON messages (1)	1380 bytes
IBMBETP	ON messages (2)	760 bytes
IBMBETQ	ON messages (3)	1020 bytes
IBMBETT	EVENT messages	1140 bytes
IBMDEDO	Open diagnostic file	210 bytes
IBMDEDW	Console transmitter	190 bytes
IBMDESM	Error message module phase 1	1010 bytes
IBMDESN	Error message module pahse 2	2250 bytes
IBMDESY	Error system action	180 bytes
IBMDKDD	Hexadecimal dump	1000 bytes
IBMDKFA	Dump file attributes	3300 bytes
IBMDKDR	Dump storage report	1100 bytes
IBMDKDT	Dump/COUNT transmitter	1550 bytes
IBMDKMR	Dump control	1040 bytes
IBMDKPT	Dump parameter translate	380 bytes
IBMDKTB	Save Area Control Block printout	2100 bytes
IBMDKTC	Save Area chain validity checker	360 bytes
IBMDKTR	Dump trace	3600 bytes
IBMDOCA	Close	1740 bytes
IBMDOCV	Close VSAM files	460 bytes
IBMDOPM	OPEN - consecutive unbuffered files	1250 bytes
IBMDOPP	OPEN - consecutive buffered files	1000 bytes
IBMDOPQ	OPEN - consecutive buffered files (level 2)	870 bytes
IBMDOPS	OPEN - stream files	1030 bytes
IBMDOPT	OPEN - stream files (level 2)	1080 bytes
IBMDOPU	OPEN - consecutive buffered/stream files (level 3)	780 bytes
IBMDOPV	Open VSAM files	2260 bytes
IBMDOPW	OPEN - indexed files (level 4)	740 bytes
IBMDOPX	OPEN - regional and indexed files	1130 bytes
IBMDOPY	OPEN - regional/indexed files (level 2)	760 bytes
IBMDOPZ	OPEN - regional indexed files (level 3)	550 bytes
IBMDPEP	Housekeeping Diagnostic message module	780 bytes
IBMDPES	Storage management Diagnostic message module	1580 bytes
IBMDPIF	Operation Exception checking, (no floating-point hardware)	130 bytes
IBMDPII	Program ISA initialization	910 bytes
IBMDPJI	Program ISA initialization from caller	760 bytes
IBMDRAW	Regional(3) sequential unbuffered output transmitter	710 bytes
IBMDRAY	Regional(3) sequential buffered output transmitter	650 bytes
IBMDRAY	Regional(1) sequential unbuffered output transmitter	770 bytes
IBMDRAZ	Regional(1) sequential buffered output transmitter	680 bytes
IBMDRBW	Regional(1) sequential buffered input/update transmitter	860 bytes
IBMDRBX	Regional(3) sequential buffered input/update transmitter	860 bytes
IBMDRBY	Regional(3) sequential unbuffered input/update transmitter	1040 bytes
IBMDRBZ	Regional(1) sequential unbuffered input/update transmitter	1020 bytes
IBMDRCY	Consecutive sequential unbuffered transmitter, U-format	1020 bytes
IBMDRCZ	Consecutive sequential unbuffered transmitter, F-format	1030 bytes
IBMDRDY	Regional(3) direct transmitter	950 bytes
IBMDRDZ	Regional(1) direct transmitter	850 bytes
IBMDREF	ENDFILE module	170 bytes
IBMDREX	Error handler for indexed files	580 bytes
IBMDREV	Error handler for VSAM files	730 bytes
IBMDREY	Error handler for regional and unbuffered consecutive files	690 bytes
IBMDREZ	Error handler for buffered consecutive files	490 bytes
IBMDRJZ	Indexed sequential input/update transmitter	1410 bytes
IBMDRKZ	Indexed direct input/update transmitter	960 bytes
IBMDRLZ	Indexed sequential output transmitter	660 bytes
IBMDRRR	Consecutive buffered exit module	300 bytes
IBMDRRT	Consecutive sequential buffered OMR transmitter, F-format	400 bytes
IBMDRRU	Consecutive sequential buffered associate files, U-format	360 bytes
IBMDRRV	Consecutive sequential buffered associate files, V-format	380 bytes
IBMDRRW	Consecutive sequential buffered associate files, F-format	620 bytes
IBMDRVR	VSAM KSDS direct transmitter	1380 bytes
IBMDRVS	VSAM KSDS sequential input/update transmitter	1820 bytes
IBMDRVT	VSAM KSDS sequential output transmitter	970 bytes
IBMDRVZ	VSAM ESDS transmitter	1460 bytes
IBMDRRX	Consecutive sequential buffered transmitter, U-format	520 bytes
IBMDRRY	Consecutive sequential buffered transmitter, V-format	620 bytes

IBMDRRZ	Consecutive sequential buffered transmitter, F-format	620 bytes
IEMISCT	Conversion conditicn interface	470 bytes
IBMDSOF	Stream output transmitter, F-format	210 bytes
IBMDSOU	Stream output transmitter, U-format	170 bytes
IBMDSOV	Stream output transmitter, V-format	210 bytes
IBMBSTA	Tab table	40 bytes
IBMDSTU	Stream print transmitter, U-format	410 bytes
IBMDSTV	Stream print transmitter, V-format	430 bytes

Index

DOS
PL/I Oprimizing Compiler:
Execution Logic

Order No. SC33-0019-1

READER'S COMMENT FORM

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

Cut or Fold Along Line

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.

Cut or Fold Along Line

Fold Fold

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE:

Fold Fold

IBM

DOS PL/I Optimizing Compiler: Execution Logic (File No. S360/S370-29) Printed in U.S.A. SC33-0019-1

SC33-0019-1

DOS PL/I Optimizing Compiler: Execution Logic (File No. S360/S370-29) Printed in U.S.A. SC33-0019-1

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, N.Y. 10604

IBM World Trade Americas/Far East Corporation
Town of Mount Pleasant, Route 9, North Tarrytown, N.Y., U.S.A. 10591

IBM World Trade Europe/Middle East/Africa Corporation
360 Hamilton Avenue, White Plains, N.Y., U.S.A. 10601