Program Product

OS PL/I Optimizing Compiler: Execution Logic

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

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

Release 5 (5.0 and 5.1)

IBM

Fourth Edition (September 1985)

This is a major revision of SC33-0025-2, which incorporates technical newsletters SN26-8171 and SN33-6173. This edition applies to Release 5.1 of:

    OS PL/I Optimizing Compiler, Program Product 5734-PL1,
    OS PL/I Resident Library, Program Product 5734-LM4,
    OS PL/I Transient Library, Program Product 5734-LM5,
    Composite package, Program Product 5734-PL3,

and to any subsequent version, release, and modification until otherwise indicated in any new editions or technical newsletters.

Specific changes are indicated by a vertical bar to the left of the change. These bars will be deleted at any subsequent republication of the page affected. Editorial changes that have no technical significance are not noted.

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

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM program product in this publication is not intended to state or imply that only IBM's program product may be used. Any functionally equivalent program may be used instead.

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

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

PREFACE

The main purpose of this publication is to explain, in general
terms, the way in which programs compiled by the O/S PL/I
Optimizing Compiler (Program Number 5734-PL1) are executed. It
describes the organization of object programs produced by the
compiler, the contents of the load module, and the main storage
situation throughout execution. The information provided is
intended primarily for those involved in maintenance of the
compiler and its related library program products. The
publication will also provide valuable information for
application programmers, because a knowledge of the way in which
source program statements are executed will lead to the writing
of more efficient programs. The book also 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 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 load module. 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 Operating System routines.
Chapter 12 is the guide to obtaining and using dumps. Chapter
13 deals with interlanguage communication. The final chapter,
Chapter 14, discusses those aspects of execution that apply only
to a multitasking environment. In addition, Appendix A 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 System/370
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

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

System/370 Principles of Operation, GA22-7000

REFERENCE PUBLICATIONS

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

OS PL/I Optimizing Compiler: Programmer's Guide, SC33-0006

OS PL/I Optimizing Compiler: Program Logic, LY33-6007

OS PL/I Optimizing Compiler: Installation, SC33-0026
(Release 4.0 only)

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

OS PL/I Optimizing Compiler: Installation for CMS, SC26-4122

OS PL/I Resident Library: Program Logic, LY33-6008

OS PL/I Transient Library: Program Logic, LY33-6009

MVS/Extended Architecture Data Management Macro
Instructions, GC26-4014

OS/VS2 MVS Data Management Macro Instructions, GC26-3873

IBM System/370 Reference Summary, GX20-1850

OS/VS2 System Programming Library: MVS Diagnostic Techniques, GC28-0725

MVS/Extended Architecture Linkage Editor and Loader, GC26-4011

OS/VS Linkage Editor and Loader, GC26-3813

CONTENTS

FIGURES

CHAPTER 1.   INTRODUCTION

PROCESSING A PL/I PROGRAM

Figure 1 on page 2 shows the process 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 (that is, creating an object module).

3.   Link-editing: producing a load module from the object module.   This includes linking the compiled code with PL/I library modules, and possibly with other compiled programs. It also includes resolving the addresses within the code.

4.   Execution: running the load module.

The process is not necessarily continuous.   The program may, for example, be kept in a compiled or link-edited form before it is executed, and it will 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 variables into a series of machine instruction.   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 OS 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 OS 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);
```

PREPARE

```
          ┌─────────────┐
          │   Source    │
          │   Program   │
          └─────────────┘
                 V
```

COMPILE

```
          ┌─────────────┐
          │    PL/I     │
          │ Optimizing  │
          │  Compiler   │
          └─────────────┘
                 V
```

LINK-EDIT

```
          ┌─────────────┐
          │   Object    │
          │   module    │
          └─────────────┘


┌───────────────┐                    ┌───────────────┐
│  PL/I library │                    │    Other      │
│  modules      │                    │    object     │
│               │                    │    modules    │
└───────────────┘                    └───────────────┘
          V         V         V
          ┌─────────────┐
          │    Load     │
          │   module    │
          └─────────────┘
                 V
```

EXECUTE

Initialization routines	Object program	Termination routine
Receive control from system, and sets up PL/I environment	Carries out actions specified in source program	Closes any files still open, and returns control to system

Figure 1.   The Process of Running a PL/I Program

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

```
LA    14,92(0,13)    Place in address of Y in register 14

LA    15,96(0,13)    Place in 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 IBMBMGSA.
                     (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, and possibly, modules produced by further compilations. As well as linking the external modules, the linkage editor also resolves addresses within the object module.

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 know 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 IBMBPIRA and IBMBPIIA.

2. Executing the program.

3. Completing the job after execution. This consists of closing any files that are left open and returning control either to the supervisor or to a calling module. It is handled by a return to the initialization routine which calls a termination routine.

FACTORS AFFECTING IMPLEMENTATION

Three major factors influence 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 the 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 reuse of storage to reduce space requirements.

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

    The PL/I language offers more facilities than most high-level languages. 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:

* A communications area addressed by a dedicated register
  throughout the execution of the program.

* A scheme to handle dynamic storage allocation.

* The use of standard subroutines from the PL/I libraries, to
  handle such standard tasks as the housekeeping scheme and
  error handling.

* The use of initialization routines to set up the
  communications area and initiate the housekeeping scheme.
  All PL/I modules are compiled on the assumption that the
  initialization routines have been called before they are
  entered.

* The issuing, by the initialization routines, of SPIE/ESPIE
  and STAE/ESTAE macro instructions to trap interrupts and
  ABENDs, and allow them to be handled as defined by PL/I.

These features are discussed further below.

COMMUNICATIONS AREA

The facilities offered by 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.

DYNAMIC STORAGE ALLOCATION

The principles of the dynamic storage scheme are illustrated in
Figure 2 on page 5.

The allocation and freeing of automatic storage on a
block-by-block basis implies an automatic facility for the reuse
of such storage. This problem and the problem of inter-block
communications 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 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 an entry to a block, the registers of the preceding block are
stored in the previous DSA and a new DSA is acquired. A
back-chain 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 the prolog code. 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.

```
┌─┬──────────────┐        ┌─┬───────────────┐        ┌─┬───────────────┐
│ │              │        │ │ Program       │        │ │ Program       │
│ │              │        │ │ mgmt area     │        │ │ mgmt area     │
Initial          │        │ │               │        │ ├───────────────┤
storage          │    ISA │ │               │    ISA │ │               │
area             │        │ │               │        │ │ LIFO storage  │
(ISA)            │        │ │               │        │ │               │
│ │              │        │ │               │        │ ├───────────────┤
│ │              │        │ │               │        │ │ Major free    │
│ │              │        │ │               │        │ │ area          │
└─┴──────────────┘        └─┴───────────────┘        └─┴───────────────┘
```

1 The initial
  storage area
  (ISA) is acquired

2 The program
  management area
  (a PL/I communications
  area) is place at the
  head of the ISA.

3 All storage freed on
  a last in/first out
  basis (LIFO storage)
  is allocated at the
  low address end of
  the remaining unused
  storage.

```
┌─┬──────────────┐        ┌─┬───────────────┐        ┌─┬───────────────┐
│ │ Program      │        │ │ Program       │        │ │ Program       │
│ │ mgmt area    │        │ │ mgmt area     │        │ │ mgmt area     │
│ ├──────────────┤        │ ├───────────────┤        │ ├───────────────┤
│ │ LIFO storage │        │ │ LIFO storage  │        │ │ LIFO storage  │
│ ├──────────────┤        │ ├───────────────┤        │ ├───────────────┤
ISA│ │            │    ISA │ │ Major free    │    ISA │ │ Major free    │
│ │              │        │ │ area          │        │ │ area          │
│ │ Major free   │        │ ├───────────────┤        │ ├───────────────┤
│ │ area         │        │ │               │        │ │ Non-LIFO storage│
│ │              │        │ │ Non-LIFO      │        │ ├───────────────┤
│ │              │        │ │ storage       │        │ │ Freed         │
│ │              │        │ │               │        │ │ non-LIFO storage│
└─┴──────────────┘        └─┴───────────────┘        └─┴───────────────┘
```

4 When LIFO storage
  is freed, the most
  recently allocated
  element is the first
  to be freed. It is
  freed by being
  reabsorbed into the
  major free area.

5 Elements not
  freed on a last in/
  first out basis
  (non-LIFO storage)
  are allocated at
  the high address end
  of the free storage.

6 When non-LIFO storage
  is freed, it is,
  where possible,
  absorbed into the
  major free area.
  Where this is not
  possible it is placed
  on a chain of free
  storage. The head of
  this chain is held at
  a fixed offset in the
  program management
  area. Areas on this
  chain are reused
  where possible.

Figure 2. Use of PL/I Dynamic Storage without Heap Storage

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 called non-LIFO storage. This storage is sometimes also
called heap. If items within this stack are freed, they are
placed on a free-area chain. The storage scheme is handled
partly by a compiled code and partly by a resident library
routine. Compiled code acquires and frees space in the LIFO
storage stack.

The library routine IBMBPGR is called when:

• Non-LIFO dynamic storage has to be allocated or freed

• There is insufficient space for an allocation of storage in
  the LIFO stack

• Additional calls overflow

USE OF LIBRARY SUBROUTINES

The use of library subroutines simplifies compilation. On the
other hand, 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 I/O error handling, and
for those items that 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 OS PL/I Optimizing Compiler:
the OS PL/I Resident Library and OS PL/I Transient Library.
Transient Library routines have the advantage of saving space,
because they require storage only when they are actually in use.
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/TERMINATION ROUTINES

The job of initialization is to prepare a standard environment
for all procedures compiled by the PL/I Optimizing Compiler.
This consists of setting up the TCA and initializing the storage
scheme. A SPIE/ESPIE macro instruction is issued so that all
the program checks will be intercepted by the PL/I error
handling facilities. A STAE/ESTAE macro instruction is issued
to trap ABENDs. On completion of the main procedure control is
returned to initialization routine by the epilog code of the
main procedure. The program is terminated under the control of
the initialization routine. 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 subroutines
can be compiled and tested individually and then joined with
other procedures and run without recompilation. If this is
done, care must be taken that the main procedure is the first
passed to the linkage editor.

Note: Use of the linkage editor ENTRY statement will not have
the desired results as the program must be entered via one of
the initialization routines.

CONTENTS OF A TYPICAL LOAD MODULE

The contents of a typical load module are shown in Figure 3 on page 8. The contents are:

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

- Link-edited routines. These routines include resident library routines, many of which are included in every executable program phase. These are the initialization routine, IBMBPIR, and the error handler, IBMBERR. Other resident routines are included as required.

As well as the executable machine instructions, the program requires certain control information and addresses. Some of these are listed in Figure 3, but the full details are given in Chapter 2, "Compiler Output" on page 12. The figure also shows 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 4 on page 9. As can be seen, an area known as the initial storage area (ISA) is acquired for the program management and PL/I dynamic storage. The program management area is set up by the initialization routines, and includes the TCA and the dummy DSA discussed above. The remainder of the ISA is used for PL/I dynamic storage allocations. The LIFO stack starts beyond the end of the program management area and expands, as necessary, toward the end of the ISA. Storage for I/O buffers and transient library routines is acquired by issuing GETMAIN macro instructions.

Non-LIFO dynamic storage may start at the end of the ISA and expand toward the LIFO stack. If heap storage is used, separate storage areas may be obtained.

PLISTART

PLIMAIN

LOAD MODULE

```
+------------------------------------------+  <---------------------+
| ADDRESSES                                |                        |
|    Addresses of:                         |                        |
|    Library modules,                      |                        |
|    PL/I subroutines and                  |                        |
|    entry points,                         |                        |
|    Label constants,                      |                        |
|    External procedures, etc.             |                        |
+------------------------------------------+                        |
| CONTROL BLOCKS                           |             Static     |
|    Various control blocks                |             internal   |
|    needed during                         |             control    |
|    execution                             |             section    |
+------------------------------------------+                        |
| CONSTANTS                                |                        |
|    Storage for any constants             |                        |
|    used in the program                   |                        |
+------------------------------------------+                        |
| STATIC VARIABLES                         |                        |
|    Storage for variables                 |                        |
|    declared as STATIC                    |                        |
|    INTERNAL                              |                        |
+------------------------------------------+  <---------------------+
| OTHER CONTROL SECTIONS                   |
|    Storage for variables                 |
|    declared as STATIC EXTERNAL.          |
|    Control blocks and                    |
|    other data for                        |
|    external files, etc.                  |
+------------------------------------------+
| PROGRAM CONTROL SECTION                  |
|    Compiled code                         |
+------------------------------------------+
| LIBRARY MODULES                          |
|    Link-edited library                   |
|    modules, including:                   |
|    IBMBPIR, and IBMBERR                   |
+------------------------------------------+
```

Figure 3.   Contents of a Typical Load Module

```
                    ┌──────────────────────────────────┐      ┌─────────┐
                    │  LOAD MODULE                     │      │         │
                    ├──────────────────────────────────┤      │         │
                    │  Compiled code                   │      │ Load Moule
                    │  Library modules                 │      │         │
                    │  Addresses                       │      │         │
                    │  Control blocks                  │      │         │
                    │  Constants                       │      │         │
                    │  Static variables                │      └─────────┘
                    ├──────────────────────────────────┤      ┌─────────┐
                    │  PROGRAM                         │      │         │
                    │  MANAGEMENT AREA                 │      │         │
                    ├──────────────────────────────────┤      │         │
 Other storage      │  TCA (task communications area)  │      │         │
 obtained by issuing│  Dummy DSA (dynamic storage area)│      │         │
 GETMAIN macros     │  Other housekeeping control blocks│     │         │
                    ├──────────────────────────────────┤      │         │
 ┌────────────────┐ │  LAST-IN/FIRST-OUT               │      │         │
 │Storage for:    │ │  (LIFO) STORAGE                  │      │         │
 │ Transient library├────────────────────────────────┤      │         │
 │ routines I/O   │ │  DSAs and VDAs (variable data areas).│  │         │
 │ buffers        │ │  Storage for automatic variables and │  │         │
 │                │ │  compiler-generated temporaries, and │  │ Initil  │
 │Plus:           │ │  other items allocated and freed on  │  │ Storae  │
 │ Further allocation│ a block and procedure basis     │      │ Area    │
 │ of dynamic storage├──────────────────────────────────┤    │ (ISA)   │
 │ if required    │ │                                  │      │         │
 └────────────────┘ │  MAJOR FREE AREA                 │      │         │
                    │                                  │      │         │
 Other storage for  │                                  │      │         │
 HEAP (if used)     │                                  │      │         │
                    ├──────────────────────────────────┤      │         │
 ┌────────────────┐ │                                  │      │         │
 │Storage for     │ │  NON-LIFO STORAGE                │      │         │
 │CONTROLLED and  │ ├──────────────────────────────────┤      │         │
 │dynamically     │ │  Storage for CONTROLLED and BASED │      │         │
 │allocated BASED │ │  variables if HEAP is not used,   │      │         │
 │variables, if   │ │  and for PL/I Library routines    │      │         │
 │HEAP is used    │ │                                  │      │         │
 └────────────────┘ └──────────────────────────────────┘      └─────────┘
```

Figure 4. Use of Storage

THE PROCESS OF EXECUTION

The process of execution is illustrated in Figure 5. The processes involved for a sample program are described below.

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

PLISTART	Initialization routines
Receives control from system	Set up TCA, initialize storage and issue SPIE/ESPIE & STAE/ESTAE to initialize PL/I error-handling scheme. Pass control to the address in PLIMAIN.

Prolog code	Functional code
Acquires DSA for main procedure, initializes control blocks, etc.	Carries out function required in source program. This usually involves calls to library subroutines

Epilog code	Termination routines
Restores IBMBPIR's registers	Close any files still open and return control to the operating system

Figure 5.   Flow of Control During Execution

During execution:

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, IBMBPIR.

3. IBMBPIR and IBMBPII (called by IBMBPIR) set up the PL/I environment. IBMBPIR 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 IBMBPIR passes control is held in the control section PLIMAIN.

4. Compiled code prolog stores the contents of the registers used by IBMBPIR 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 data management routines.

6. Processing is 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 epilog code is entered. This restores the registers of IBMBPIR, the initialization routine, and returns control to IBMBPIR.

9. IBMBPIR may raise the FINISH condition, calling the resident error-handling module IBMBERR, when a FINISH ON-unit is used in the main procedure. Otherwise, IBMBPIR calls IBMBPIT to carry out certain housekeeping tasks, including calling entry point IBMBOCLB in module IBMBOCL to close files. IBMBPIT returns control to IBMBPIR; if user-exit IBMBEER is present PL/I either ABENDs or returns control to the calling program.

This program illustrates the main points mentioned earlier in the chapter. The initialization routines are used in steps 3 and 9, to set up and discard the PL/I environment. The storage environment scheme is illustrated in the prolog and epilog code in steps 4 and 8. The communications area (TCA) is set up by the initialization routine, and the use of 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

This chapter describes the part of the load module that is generated by the compiler. 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
> Contain machine instructions or constants.

RLD records
> Contain internal addresses that require updating for a load module.

ESD records
> Contain external names to be resolved (bound) with other programs and data areas.

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

There are two main control sections produced by the compiler. These are:

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

• 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 by other compilations.

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

The output from the compiler is shown in Figure 6 on page 13 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.

Program control section

```
┌──────────────────────┐                ┌─────────────────────────────┐
│                      │                │ Contains:                   │
│                      │ ───>           │                             │
│                      │ ───>           │ Executable instructions     │
│                      │                │ translated from source      │
│                      │                │ program                     │
│                      │                └─────────────────────────────┘
│                      │
│                      │                Static internal
│                      │                control section
│      COMPILER        │                ┌─────────────────────────────┐
│                      │                │ Contains:                   │
│                      │                │                             │
│                      │ ───>           │ Addresses                   │
│                      │ ───>           │ Constants                   │
│                      │                │ Control information         │
│                      │                │ Static internal             │
│                      │                │ variables                   │
└──────────────────────┘                └─────────────────────────────┘
```

Housekeeping
control sections

```
┌──────────────────────┐                Control sections for
│ PLISTART             │                data declared
│ Contains:            │                EXTERNAL
│   Instructions       │                ┌─────────────────────────────┐
│   passing control    │                │ A separate control section  │
│   to initialization  │                │ for each external:          │
│   routine            │                │   Variable                  │
├──────────────────────┤                │   File                      │
│ PLIMAIN              │                │   Procedure                 │
│ Contains:            │                │   User condition            │
│   Address of main    │                │   Symbol table for external data │
│   procedure          │                └─────────────────────────────┘
├─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ┤
│ PLIFLOW              │
│ Contains:            │                Dummy Section  V
│   External reference │                ┌ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ┐
│   to library module  │                │ A dummy section      │
│   used in FLOW       │                │ containing address   │
│   option             │                │ information for      │
├─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ┤                │ file and controlled  │
│ PLICOUNT             │                │ variables.           │
│ Contains:            │                │                      │
│   External reference │                │ Becomes the          │
│   to library module  │                │ pseudo-register      │
│   used in COUNT      │                │ vector (PRV)         │
│   option             │                └ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ┘
└──────────────────────┘
```

```
┌ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ┐
│ A control section for│
│ compiler-generated   │
│ subroutine           │
└ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ┘
```

Note: Control sections surrounded with broken lines are generatedonly when required.

Figure 6. The Output from the Compiler

2. <u>Control sections that are generated only when required</u>

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 COUNT
           compiler option is specified.

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.

3.   Dummy sections

     Pseudo-register vector
           A dummy section used in addressing files and
           controlled variables.

The two control sections, PLISTART and PLIMAIN, are used during
program initialization. PLISTART holds the address of the
library initialization routine IBMBPIR, 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.

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

The format of PLIMAIN and PLISTART is given in
Appendix A, "Control Blocks" on page 326.

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 IBMBEFL and also
contains the values of "n" and "m" specified in the option. The
format of PLIFLOW is given in Chapter 7, "Error and Condition
Handling" on page 105.

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 7 shows all the 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 installation time. To obtain deleted options, the
correct password (specified at installation 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 OS PL/I Optimizing Compiler:
Programmer's Guide.

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 references generated by the compiler[1]	ESD
Static storage	Contents of static internal and static external control sections in hexadecimal notation with comments	MAP and LIST
Table of offset and statement number	Offsets, within code, of the start of each statement	OFFSET
Object program	The contents of the program control section in hexadecimal and translated into a pseudo-assembler-language format	LIST
Variables offset MAP	The offsets of automatic and static internal variables from their defining base	MAP

Figure 7.   Contents of Listing and Associated Compiler Options

Note to Figure 7:

[1]   External references within library modules are not included.

STATIC-STORAGE MAP

The static-storage map is a formatted listing of the contents of the static internal and static external control sections. You obtain this listing by specifying the MAP option in the PROCESS statement. 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, declare control blocks (DCLCBs), 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 static listing is shown in Figure 8 on page 17.

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

A..   Address constant

COMPILER LABEL CL.nn
                        Compiler-generated label followed by CL plus
                        number

CONDITION CSECT   Control section for programmer-named condition

CONSTANT

CSECT FOR EXTERNAL VARIABLE
                        Control section for external variable

D..   Descriptor

DED..   Data element descriptor

ENVB   Environment control block

DCLCB   Declare 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 for xxx

xxx   Name of 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.)

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

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

       4   1                     DO I=1 TO Y;
       5   1   1                 Z=Z*X(I);
       6   1   1                 END;
       7   1              PUT EDIT(Z)(A);
       8   1              END;
```

	STATIC INTERNAL STORAGE MAP			STATIC EXTERNAL CSECTS	
000000	E00000F8	PROGRAM ADCON	000000	0000000000000000	DCLCB
000004	00000008	PROGRAM ADCON		0000000000000000	
000008	0000005E	PROGRAM ADCON		000000140005E2E8	
00000C	00000068	PROGRAM ADCON		E2C9D500	
000010	00000068	PROGRAM ADCON			
000014	00000000	A...IELCGIX			
000018	00000000	A...IELCGIB	000000	FFFFFFFC41201000	DCLCB
00001C	00000000	A...IBMBCACA		02D70F0000000000	
000020	00000000	A...IBMBCEDB		000000140008E2E8	
000024	00000000	A...IBMBCHFD		E2D7D9C9D5E30000	
000028	00000000	A...IBMBCTHD			
00002C	00000000	A...IBMBCVDY			
000030	00000000	A...IBMBOCLA			
000034	00000000	A...IBMBOCLC			
000038	00000000	A...IBMBSAOA			
00003C	00000000	A...IBMBSEDB			
000040	00000000	A...IBMBSEIA			
000044	00000000	A...IBMBSEIT			
000048	00000000	A...IBMBSFIA			
00004C	00000000	A...IBMBSIIA			
000050	00000000	A...IBMBSIOA			
000054	00000000	A...IBMBSIOT			
000058	00000000	A...IBMBSXCA			
00005C	00000000	A...STATIC			
000060	08040680	DED..X			
000064	500000030080	FED			
00006A	6000000B	FED			
00006E	58010000	FED			
000072	000A	CONSTANT			
000074	0001	CONSTANT			
000076	0004	CONSTANT			
000078	91E091E0	CONSTANT			
00007C	00000000	CONSTANT			
000080	46008000	CONSTANT			
000084	00000000	A..DCLCB			
000088	00000000	A..DCLCB			
00008C	00000000	A..DCLCB			
000090	80000000	A..TEMP			
000094	00000000	A..DCLCB			
000098	80000000	A..TEMP			
00009C					
0000A0	0000010600000068	COMPILER LABEL CL.11			

Figure 8. Example of Static Storage Listing

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. This listing consists of the machine instructions plus a translation of these instructions into a form that resembles assembler language, and number of comments such as the statement number. The format of this listing is shown in Figure 9 on page 19. 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 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 prolog, which carries out various housekeeping tasks and is described more fully later in this chapter. The end of the prolog 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.

- PROLOG BASE—identifies the start of the prolog 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 the program size exceeds 4096 bytes and consequently cannot be addressed from one base.

- CONTINUATION OF PREVIOUS REGION—identifies 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.

```
00004A  91 40 F 02C        TM    44(15),X'40'
00004E  47 80 7 056        BZ    *+8
000052  96 80 1 010        OI    16(1),X'80'
000056  48 50 F 050        LH    5,80(0,15)
00005A  4B 50 E 002        SH    5,2(0,14)
00005E  91 C0 E 001        TM    1(14),X'C0'
000062  47 E0 7 076        BNO   *+20
000066  4B 50 E 002        SH    5,2(0,14)
00006A  91 40 F 026        TM    38(15),X'40'
00006E  47 80 7 076        BZ    *+8
000072  06 50              BCTR  5,0
000074  06 50              BCTR  5,0
000076  40 50 F 050        STH   5,80(0,15)
00007A  58 50 F 04C        L     5,76(0,15)
00007E  50 50 1 000        ST    5,0(0,1)
000082  4A 50 E 002        AH    5,2(0,14)
000086  91 C0 E 001        TM    1(14),X'C0'
00008A  47 E0 7 09E        BNO   *+20
00008E  4A 50 E 002        AH    5,2(0,14)
000092  91 40 F 026        TM    38(15),X'40'
000096  47 80 7 09E        BZ    *+8
00009A  41 55 0 002        LA    5,2(5,0)
00009E  50 50 F 04C        ST    5,76(0,15)
0000A2  58 50 1 01C        L     5,28(0,1)
0000A6  D2 03 1 01C D 04C  MVC   28(4,1),76(13)
0000AC  07 F6              BR    6
0000AE  58 F0 7 0CC        L     15,204(0,7)
0000B2  95 60 E 000        CLI   0(14),X'60'
0000B6  47 70 7 0BE        BNE   *+8
0000BA  58 F0 7 0D0        L     15,208(0,7)
```

1PL/I OPTIMIZING COMPILER   EXAMPLE: PROC OPTIONS(MAIN) REORDER;   PAGE 8

```
-00003C  18 F0              LR    15,0
00003E   90 E0 1 048        STM   14,0,72(1)
000042   50 D0 1 004        ST    13,4(0,1)
000046   41 D1 0 000        LA    13,0(1,0)
00004A   50 50 D 058        ST    5,88(0,13)
00004E   92 80 D 000        MVI   0(13),X'80'
000052   92 24 D 001        MVI   1(13),X'24'
000056   D2 03 D 054 3 078  MVC   84(4,13),120(3)
00005C   05 20              BALR  2,0
```

* PROLOGUE BASE

* INITIALIZATION CODE FOR Z
```
00005E   78 40 3 07C        LE    4,124(0,3)
000062   70 40 D 0BC        STE   4,Z
```
* END OF INITIALIZATION CODE FOR Z

```
000066   05 20              BALR  2,0
```

* PROCEDURE BASE

* STATEMENT NUMBER 3
```
000068   41 70 D 100        LA    7,256(0,13)
00006C   50 70 3 090        ST    7,144(0,3)
000070   96 80 3 090        OI    144(3),X'80'
000074   41 10 D 100        LA    1,256(0,13)
000078   50 10 D 0F0        ST    1,240(0,13)
00007C   92 24 D 111        MVI   273(13),X'24'
000080   41 E0 3 0A0        LA    14,160(0,3)
000084   50 E0 D 118        ST    14,280(0,13)
000088   41 10 3 08C        LA    1,140(0,3)
```

```
000030   58 F0 7 03C              L     15,60(0,7)
000034   18 E6                    LR    14,6
000036   07 FF                    BR    15
000038                            DC    AL4(0)
00003C                            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 014              B     *+16
000010   00000000                 DC    A(STMT. NO. TABLE)
000014   00000130                 DC    F'304'
000018   00000000                 DC    A(STATIC CSECT)
00001C   58 30 F 010              L     3,16(0,15)
000020   58 10 D 04C              L     1,76(0,13)
000024   58 00 F 00C              L     0,12(0,15)
000028   1E 01                    ALR   0,1
00002A   55 00 C 00C              CL    0,12(0,12)
00002E   47 D0 F 030              BNH   *+10
000032   58 F0 C 074              L     15,116(0,12)
000036   05 EF                    BALR  14,15
000038   58 E0 D 048              L     14,72(0,13)
0000D2   05 AA                    BALR  10,10
0000D4   47 F0 2 09E              B     CL.11
0000D8              CL.10         EQU   *
0000D8   41 E0 3 064              LA    14,100(0,3)
0000DC   58 10 D 0F0              L     1,240(0,13)
0000E0   58 70 3 014              L     7,A..IELCGIX
0000E4   05 67                    BALR  6,7
0000E6   58 F0 3 048              L     15,A..IBMBSFIA
0000EA   05 EF                    BALR  14,15
0000EC   58 70 3 018              L     7,A..IELCGIB
0000F0   05 67                    BALR  6,7
0000F2   05 AA                    BALR  10,10
0000F4   41 E0 3 06A              LA    14,106(0,3)
0000F8   58 10 D 0F0              L     1,240(0,13)
0000FC   58 70 3 014              L     7,A..IELCGIX
000100   05 67                    BALR  6,7
000102   47 F0 2 070              B     CL.10
000106              CL.11         EQU   *
```

* STATEMENT NUMBER 4
```
000106   78 00 D 0B8              LE    0,Y
00010A   70 00 D 0F8              STE   0,248(0,13)
00010E   48 70 3 074              LH    7,116(0,3)
000112   40 70 D 0C0              STH   7,I
000116   48 40 D 0C0              LH    4,I
00011A   50 40 D 128              ST    4,296(0,13)
00011E   48 40 3 080              LH    4,128(0,3)
000122   40 40 D 128              STH   4,296(0,13)
000126   97 80 D 12A              XI    298(13),X'80'
00012A   78 20 D 128              LE    2,296(0,13)
00012E   7B 20 3 080              SE    2,128(0,3)
```

```
00008C   58 F0 3 04C        L     15,A..IBMBSIIA
000090   05 EF              BALR  14,15
000092   41 A0 2 070        LA    10,CL.10
000096   48 50 3 074        LH    5,116(0,3)
00009A   50 50 D 0F4        ST    5,244(0,13)
00009E              CL.5    EQU   *
00009E   18 45              LR    4,5
0000A0   8B 40 0 002        SLA   4,2
0000A4   41 E4 D 0C4        LA    14,VO..X(4)
0000A8   41 F0 3 060        LA    15,DED..VO..X
0000AC   58 10 D 0F0        L     1,240(0,13)
0000B0   90 EF 1 008        STM   14,15,8(1)
0000B4   05 AA              BALR  10,10
0000B6   4A 50 3 074        AH    5,116(0,3)
0000BA   50 50 D 0F4        ST    5,244(0,13)
0000BE   49 50 3 072        CH    5,114(0,3)
0000C2   47 C0 2 036        BNH   CL.5
0000C6   41 E0 D 0B8        LA    14,Y
0000CA   41 F0 3 060        LA    15,DED..Y
0000CE   90 EF 1 008        STM   14,15,8(1)
```

1PL/I OPTIMIZING COMPILER   EXAMPLE: PROC OPTIONS(MAIN) REORDER;

```
-00015C  50 70 D 128        ST    7,296(0,13)
000160   48 70 3 080        LH    7,128(0,3)
000164   40 70 D 128        STH   7,296(0,13)
000168   97 80 D 12A        XI    298(13),X'80'
00016C   78 60 D 128        LE    6,296(0,13)
000170   7B 60 3 080        SE    6,128(0,3)
000174   79 60 D 0F8        CE    6,248(0,13)
000178   47 C0 2 0D0        BNH   CL.2
```

```
000132   39 20                    CER   2,0
000134   47 20 2 114              BH    CL.3
000138              CL.2          EQU   *
```

* STATEMENT NUMBER 5
```
000138   48 90 D 0C0              LH    9,I
00013C   8B 90 0 002              SLA   9,2
000140   78 40 D 0BC              LE    4,Z
000144   7C 49 D 0C4              ME    4,VO..X(9)
000148   70 40 D 0BC              STE   4,Z
```

* STATEMENT NUMBER 6
```
00014C   48 70 D 0C0              LH    7,I
000150   4A 70 3 074              AH    7,116(0,3)
000154   40 70 D 0C0              STH   7,I
```

* CODE MOVED FROM STATEMENT NUMBER 4
```
000158   48 70 D 0C0              LH    7,I                                  PAGE  9
0001EA   07 07                    NOPR  7
```

* END PROGRAM

Figure 9. Part of an Object Program Listing

- 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. (See Chapter 13)

- 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 code that follows has been moved from within a loop by the optimization process.

- CODE MOVED FROM STATEMENT NUMBER n—indicates object code moved by the optimization process 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 an expression used more than once in the program is calculated at this point.

- 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 the name of a PL/I variable. 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 8 on page 17.

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

1. Addresses in static CSECT and code CSECT

2. Addresses of library modules

3. Addresses of entry points

4. Addresses of label constants that may be assigned to label
   variables

5. 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, "Error and Condition Handling" on page 105)
Descriptors, locators and DEDs (data element descriptors)	Control information used by compiled code and library. (See Chapter 4, "Communication between Routines" on page 64.)
Symbol table address vector	Control information used in data-directed I/O. (See Chapter 4, "Communication between Routines" on page 64.)
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 next section of the static internal control section holds
the static variables. These are held in size order, with the
smallest being first.

The final section of the static internal control section
contains branch tables for those select-groups for which
optimized code has been produced, the statement number tables
containing GOSTMT and GONUMBER data (if one of those options has
been specified) and the TIMESTAMP data (if this option has been
specified at installation time).

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.

REGISTER USAGE

Details of register usage during the execution of compiled code are given in Figure 10.

	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 size greater than 3896 bytes
5		General + static back-chain on entry to procedure	Preferred register for DO-loop control variable	
6		General		
7		General		
8		General		
9		General		
10		General	Preferred register for DO-loop control when BXLE instruction is used	
11		General	Preferred register for DO-loop control when BXLE instruction is used	
12	Address of TCA			

Figure 10 (Part 1 of 2). Register Usage in Compiled Code

	Dedicated Registers	Work Registers (plus special use)	Preferred Registers	Notes
13	Address of current DSA			
14		General + branch-and-link to library and other routines		
15		General + branch-and-link to library and other routines		

Figure 10 (Part 2 of 2).   Register Usage in Compiled Code

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

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 five even/odd work register pairs. These are (0,1), (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 10 on page 22.

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" on page 21). 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, "Object Program Initialization" on page 74. Its format is shown in Appendix A, "Control Blocks" on page 326.

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. These are discussed in Chapter 5, "Object Program Initialization" on page 74; the format of the housekeeping information in a DSA is given in Appendix A, "Control Blocks" on page 326.

REGISTER 4: When the DSA is larger than 3896 bytes register 4 is used as a base for compiler generated temporaries.

Work Registers

Special or preferred uses for work registers are shown in Figure 10 on page 22. 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 11 on page 25.

In both library and compiled code usage, register 12 points at the TCA, and register 13 at the current DSA. Registers 14 and 15 are used by both library subroutines and compiled code to branch and link between routines.

A further point about library register usage is worth noting. 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.

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 11. Library Register Usage

HANDLING AND ADDRESSING VARIABLES AND TEMPORARIES

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, 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 prolog code after the DSA has been acquired. The same method is used as is used for acquiring the DSA, as described in "Prolog" on page 32.

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.

Automatic variables known in any block are those that are declared in that block, or in any encompassing blocks. The method used to address automatic variables in outer blocks is a static back-chain.

The compiler-generated prolog for a procedure saves the address of the static back-chain DSA. This address can then be accessed from register 13. Frequently, the value is retained in the

register 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 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. Temporaries are addressed from register
13, unless the DSA is longer than 4096 bytes. Because temporary
storage is continually being reused, the same offset will not
always refer to 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 the pseudo-register
vector, as described below under "The Pseudo-register Vector
(PRV)" on page 27. When no allocations of the controlled
variable have been made, the PRV offset points to the dummy FCB.
Otherwise, it points to the most recent allocation of the
controlled variable.

Each controlled variable is headed by a four-word control block
that holds the address of the previous allocation (if any), the
length of the variable (including the control block), the
pseudo-register vector offset, and the task invocation count.
The format of this control block is shown in
Appendix A, "Control Blocks" on page 326.

Storage for controlled variables is allocated in non-LIFO
storage, or in separate heap storage. If there is room in the
ISA and heap is not being used, it is allocated within the ISA.
Otherwise, a GETMAIN macro instruction is issued to obtain
storage.

Stacking and unstacking of controlled variables is handled by a
resident library routine, IBMBPAFA. IBMBPAFA calls on IBMBPGR
to obtain and release the storage.

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 IBMBPGR is made. IBMBPGR acquires storage in
the heap storage area or in the ISA non-LIFO dynamic storage
area. IBMBPGR then 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 IBMBPGR is made to
free the storage.

POINTERS: Pointers and offsets 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 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 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.

THE PSEUDO-REGISTER VECTOR (PRV)

Addressing Controlled Variables and Files

In order to address controlled variables, fetched procedures, and files, PL/I uses a control block called the pseudo-register vector (PRV). This control block is mapped by the linkage editor as a dummy section with a fullword field for each uniquely named controlled variable or file. During execution, the addresses of the storage allocated to the variables, fetched procedures, or files are placed in the PRV.

For an introduction to pseudo-registers, see OS/VS Linkage Editor and Loader, or MVS/Extended Architecture Linkage Editor and Loader.

The use of the linkage editor is necessary because controlled variables and files may be external and, consequently, it may be necessary to access them in separately compiled procedures. Other external items are compiled as CSECTs, but this is not possible for files or controlled variables because their associated storage is not allocated until execution. Controlled variables have storage allocated during the execution of an

ALLOCATE statement; files are addressed from file control blocks
(FCBs), which are created when the file is opened during
execution. The use of the linkage editor means that FETCHed
procedure can not use controlled variables or files, except
SYSPRINT.

References to controlled variables and files are compiled as
assembler Q-type address constants. During link-editing, the
assembler DXD facility of the linkage editor is used, and the
PRV is set up as an external dummy section. The address of the
PRV is placed in the TCA. Each uniquely named file or
controlled variable is allocated an offset within the PRV by the
linkage editor. The Q-type address constants are then replaced
by this offset.

Controlled variables and files are addressed via the PRV
regardless of whether they are external or internal. The
compiler prefixes internal items with the name of their
procedure so that their names will be unique. The use of the
PRV is summarized in Figure 12.

During compilation

1. Each controlled variable or file reference is compiled as a
   Q-type address constant that will be used as an offset
   within the PRV.

2. The compiler generates a DXD instruction for every item
   requiring pseudo-register addressing.

During link-editing

1. The number of unique names requiring pseudo-register
   addressing is calculated and placed in a field that can be
   accessed by a CXD instruction.

2. Each reference to a name generated as a Q-type address
   constant is replaced by the appropriate offset from the
   start of the PRV.

During program initialization

1. The length required for the PRV is obtained by use of a CXD
   instruction. Storage for the PRV is then obtained in the
   program management area. The address of the PRV is placed
   in the TCA.

2. The address of the dummy FCB is placed in every field of the
   PRV.

During execution

1. When storage is allocated to the FCB or controlled variable,
   the address of the storage is placed in the associated field
   in the PRV. Comparison with the dummy FCB address can then
   be made, to determine whether storage has been allocated for
   the item.

Figure 12. Use of the Pseudo-register Vector (PRV)

The Location of the PRV

The pseudo-register vector is held in the program management area, and is addressed from the TCA.

UNDER MULTITASKING: Whenever a new task is attached, the PRV of the attaching task is copied into the program management area of the attached task. This means that, at the point when the task is attached, the files and controlled variables addressed from the subtask will be the same as those in the parent task. However, because each task has its own PRV, either task may change the addresses without affecting the other.

Initialization of the PRV

To simplify implicit opening of a record I/O file, the PRV is initialized with every field set to point to a control block known as the dummy FCB. Use of this control block as if it were a genuine FCB results in control being passed to the open routines: the file is opened, and a real FCB is created. The address of the real FCB is then placed in the PRV.

Pseudo-register fields for controlled variables are also initialized to point to the dummy FCB, so that the controlled variable allocation mechanism can determine whether an allocation has been made by comparing the PRV value with the address of the dummy FCB. (The address of the dummy FCB is held throughout the program in the TCA, so that the comparison can be made.)

PROGRAM CONTROL DATA

Program control data comprises pointer, offset, file, area, entry, event, task, and label data.

Pointer and offset data items are each held in fullword. The data item in both cases consists of an address that is held right-adjusted in the field, padded on the left with zeros. For both data types, the null value is represented by hexadecimal X'FF000000'.

A file variable is held as a fullword containing the address of the declare control block (DCLCB); the DCLCB corresponds to a file constant.

The formats of area, entry, event, task, and label data are given in Appendix A, "Control Blocks" on page 326.

HANDLING DATA AGGREGATES

PL/I data aggregates are structures and arrays, and include 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 there had been such an element 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 array descriptor during execution. Array descriptors are described in Chapter 4, "Communication between Routines" on page 64.

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, "Communication between Routines" on page 64.

ARRAYS OF STRUCTURES AND STRUCTURES OF ARRAYS

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

The array of structures

```
1 S(2),
  2 B,
  2 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

```
1 S,
  2 B(2),
  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. The difference would be in the value of the 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, "Communication between Routines" on page 64).

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 is made to it as a whole, and the elements are not 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 nonconsecutive 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 and Condition Handling" on page 105.

ACTIVATING AND TERMINATING BLOCKS

BEGIN, 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 prolog 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 prolog 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.

PROLOG AND EPILOG CODE

Except for certain single statement ON-units, every PL/I begin block or procedure block has a prolog and an epilog. The prolog prepares the environment for the associated block and acquires storage for automatic variables, compiler-generated temporaries, and workspace. The epilog frees the storage acquired for the block, restores the registers of the caller, and returns control to the caller.

Prolog

The prolog appears on the object-program listing between REAL ENTRY and PROCEDURE BASE or BLOCK BASE. Every prolog 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 prolog 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.

- Initialization of certain items for argument lists.

- Setting-up certain interrupt-handling information such as ONCBs and enable cells. (See Chapter 7, "Error and Condition Handling" on page 105).

An example of prolog code is shown in Figure 13 on page 33.

```
STM     14,12,12(13)              Store registers of calling program.
B       *+16                      Branch around constants.
DC      A(STMT. NO. TABLE)        Constant - address of statement number table.
DC      F'304'                    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 in segment.
L       15,116(0,12)              Load address of stack overflow routine
                                     (IBMBPGR) from TCA.
BALR    14,15                     Branch to overflow routine.
L       14,72(0,13)               Load address of LWS from old DSA.
LR      15,0                      Set up new NAB address.
STM     14,0,72(1)                Set LWS, NAB, and end-of-prolog NAB in
                                     DSA.
ST      13,4(0,1)                 Place back-chain in new DSA.
LA      13,0(1,0)                 Point register 13 to new DSA.
ST      5,88(0,13)                Set up static back-chain.
MVI     0(13),X'80'               Set up housekeeping flags - see
MVI     1(13),X'24'                  Appendix A.
MVC     84(4,13),120(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 13.   Typical Prolog Code

After saving the registers, the prolog tests to see if there is
enough room for the DSA in the current segment of storage. This
is done by adding the length of the new DSA, calculated at
compile time, to the address of the next available byte. For
further details on how the storage management overflow routine
operates, see "Storage Reports" on page 98.

If the result is greater than the end-of-segment pointer (EOS)
placed in the TCA during initialization, the library overflow
routine (IBMBPGR) is called to try to acquire a further segment
from the free-area chain. This process is further described in,
"Acquiring a New Segment of LIFO Storage" on page 93.

If space for the DSA is available, the next-available-byte
pointer (NAB) is updated to point at the first 8-byte boundary
beyond the end of the new DSA. The remaining instructions set
up housekeeping fields and point registers at various standard
fields, including register 13 to the start of the new DSA, and
register 4 to the start of storage for temporaries. The final
BALR instruction establishes register 2 as the program base
register.

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

Static back-chains 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 prolog is considerably different. See Chapter 13, "Interlanguage Communication" on page 281.

The format of the DSA is shown in Figure 14; full details are shown in Appendix A, "Control Blocks" on page 326.

```
R13 ──>  ┌──────────────────────────────┐
         │ Housekeeping information     │
         │ See Appendix A               │
         ├──────────────────────────────┤ ┐
         │ Items<9 bytes in length      │ │
         │                              │ │
         │ Held in alignment order:     │ │
         │       doubleword             │ │
         │       fullword               │ │
         │       halfword               │ │  Storage for automatic
         │       byte                   │ │  variables declared in
         │       bit                    │ │  the block, dynamic
         ├──────────────────────────────┤ │  ONCBs etc.
         │ Items 9-2048 bytes in length │ │
         │                              │ │
         │ Held in alignment order as   │ │
         │ above                        │ │
         ├──────────────────────────────┤ │
         │ Items>2048 bytes             │ │
         │                              │ │
         │ Held in alignment order as   │ │
         │ above                        │ │
         ├──────────────────────────────┤ ┘
         │ Parameter storage area       │ ┐
         │ Addresses of any parameters  │ │
         │ passed to the associated     │ │
         │ procedure are stored here    │ │
         ├──────────────────────────────┤ │
         │ Register bind storage area   │ │
         │                              │ │
         │ Used by compiled code when   │ │
         │ registers must be saved      │ │
         ├──────────────────────────────┤ │  Temporary storage
         │ Local temporary storage      │ │
         │                              │ │
         │ Used for temporaries required│ │
         │ for duration of statement    │ │
         ├──────────────────────────────┤ │
         │ Global temporary storage     │ │
         │                              │ │
         │ Used by temporaries required │ │
         │ for duration of block        │ │
         └──────────────────────────────┘ ┘
```

Figure 14. Contents of Typical Compiled Code DSA

Epilog

Epilog code consists of 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 epilog code is shown in Figure 15.

Epilog code for main procedure

```
LR    0,13              Save current DSA address
L     13,4(0,13)        Back-chain
L     14,12(0,13)       Pick up value of R14
LM    2,12,28(13)       Restore registers 2 through 12
BALR  1,14              Branch to initialization routine retaining
                        current address in R1
```

Epilog code for subroutine or begin block

```
L     13,4(0,13)        Back-chain
LM    14,12,12(13)      Restore registers of preceding block
BR    14                Return
```

Figure 15.   Epilog Code

The completion of a main procedure results in the raising of the FINISH condition, and this may result in the execution of an ON-unit.

Consequently, the address of the current DSA and the address of the current statement must be retained (the DSA is needed to search for the ON-unit; the address of the current statement is needed if a SNAP trace is requested in the FINISH ON-unit). Epilog code for a main procedure therefore takes a different form to that generated for a subroutine.

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 50           LR    5,13        Load static back-chain address
00031C  58 F0  3 020    L     15,A...X    Pick up address of procedure X
000320  05 FF           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. In those cases where the compiler cannot build the parameter list, the typical code might 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 56             LR   5,13              Load static back-chain 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
```

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 back-chain must be made to correct DSA. A
back-chain is made through any BEGIN blocks. The depth of
nesting can be determined during compilation, so the back-chain
can be loaded the required number of times before the branch is
made.

Typical code would be:

```
0003F0  58 D0  D 004      L    13,4(0,13)    Pick up DSA back-chain
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.

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.

This interpretive code is held in the TCA. The compiled code
branches to the interpretive code to implement a GOTO that will
or may transfer control out of the block. This TCA code
determines whether it is one of a small number of special cases,
and, if it is, calls a library routine—IBMBPGO. In other
circumstances, the GOTO code in the TCA handles the branch and
any block termination involved.

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 passes 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" on
page 105, the enablement of conditions is recorded by enable
cells. Two sets are used: the <u>block</u> enable cells retain the
enablement situation at the start of the block, which can
consequently be restored at any time; the <u>current</u> 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, 4, 12, or 13 as these are not altered
within a block. When OPTIMIZE (TIME) is specified an attempt is
made to retain other register values across labels.

Labeled statements within a block have an effect on optimization
in that, apart from the bases and block addresses mentioned
above, values cannot normally 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 restores registers 3 and 4 from the DSA passed to
it, loads register 2 from the second word of the label constant,
and loads register 13 from register 15. It 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, a search is
made for 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 left the block acquired a VDA. The
start-of-block NAB value is retained in the DSA and is known as
the end-of-prolog NAB. If a VDA has been acquired, the fact is
flagged in the flag byte of the DSA, and the GOTO places the
end-of-prolog 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 E6        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
```

Errors When Using Label Variables

Although it is invalid PL/I, it is possible for a GOTO statement
using a label variable to result in transfer of control to an
inactive block. The optimizing compiler has no method of
checking such errors, and the consequences are unpredictable.
Such errors can occur because a label variable is not reset when
the block containing the label constant to which it refers is
terminated. When an attempt is made to GOTO a label variable,
the address of the DSA is passed in register 14. The GOTO code
verifies this address to be the address of an active DSA, and
acts accordingly. Three possibilities arise:

1. The original DSA has not been overwritten, and the program
   will execute.

2. The DSA of another active block has overwritten the original
   DSA. The results are then unpredictable, as the code
   branched to will be accessing an incorrectly mapped DSA.

3. The original DSA has been overwritten with other
   information. Again, the results are not predictable. When
   PL/I determines that the data in the DSA is not another DSA,
   ERROR condition code 9002 is raised.

It should be noted that, because of the method used to allocate
DSAs, the chances of one DSA starting at the same address as a
previous DSA are high.

GOTO-Only ON-Units

As explained in Chapter 7, "Error and Condition Handling" on page 105, certain ON-units are not executed as separate program blocks. Instead, the required action is taken under the control of the error handler. ON-units containing only a GOTO statement (GOTO-only ON-units) are handled in this way.

The error handler accesses ON-units through control blocks known as ON control blocks (ONCBs). The ONCB for a GOTO-only ON-unit is specially flagged, and the last word of the ONCB is initialized to hold an offset. At this offset in the DSA of the block containing the ON-unit, the address of the label information is held. For a label variable, the offset contains the address of the label variable; for a label constant, the offset contains the address of a label temporary that is initialized to the value of the label constant. The initialization is done during the execution of the prolog of the block that contains the ON-unit.

The error handler loads the information in the label variable or the label temporary into registers 14 and 15, and calls the GOTO code in the TCA.

Interpretive GOTO Routines

If the test in the GOTO code in the TCA reveals that an abnormal situation exists, the interpretive GOTO routine is called. This routine is a subroutine of the program initialization routine.

Two abnormal cases can arise:

GOTO out of SORT exit routine

GOTO from an event I/O ON-unit (certain cases only)

When either of these situations could occur a flag is set in the TCA. Sort exits are also flagged in the DSA of the procedure involved.

The SORT exit DSA requires special action because the GOTO will involve the termination of SORT if it transfers control to another block.

The GOTO during an event I/O ON-unit can cause the termination of a number of WAIT statements. This involves removing information about these statements from the various chains that are set up during event I/O.

These two situations are explained further under the headings "SORT/MERGE" and "WAIT" in Chapter 11, "Miscellaneous Library Subroutines and System Interfaces" on page 230.

If CHECK enablement has to be changed because of a GOTO, the interpretive GOTO routine calls the library routine IBMBPGO to reset check enablement. IBMBPGO is described in the licensed publication, OS PL/I Resident Library: Program Logic.

ARGUMENT AND PARAMETER LISTS

In PL/I usage, a parameter list is a list of the items a program expects to receive; 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, "Communication between Routines" on page 64.

When arguments are passed to routines whose entry points are declared with the ASSEMBLER, COBOL, or FORTRAN attributes, the address of the data itself must be passed. The method used is described in Chapter 13, "Interlanguage Communication" on page 281.

Arguments are passed in an argument list addressed by register 1. For nonreentrant, nonrecursive code, the list is set up in static storage and completed by the compiler if the values are are known at compile time. If the procedure is reentrant, recursive, or fetched, the list is moved into the temporary storage area in the DSA before the call is made; otherwise the parameter list is moved into automatic storage.

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 13 on page 33) 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 routines 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.

Figure 16 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 "IBMB" followed by four 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.

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.

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..U(11)	Load address of argument in register
STM	14,15,0(1)	Store into argument list
L	15,A..IBMBSLO	Pick up address of routine from static internal control section and place in R15
BALR	14,15	Branch and link to routine

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

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

Figure 16. Examples of Library Calling Sequences

The fifth letter of the entry point name is mnemonic, indicating the type of module that is being called. Figure 17 gives the meaning of the mnemonics. Full details of the library modules are given in the program product publications <u>OS PL/I Transient Library: Program Logic</u> and <u>OS PL/I Resident Library: Program Logic</u>.

A further discussion of library module naming conventions is given in Chapter 3, "The PL/I Libraries" on page 53.

Mnemonics	Meaning
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 pseudo-variable routine

Figure 17.   Mnemonic Letters in Library Module Entry-Point Names

---

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 16 on page 40, 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 Subroutines

As can be seen in example 1 of Figure 16 on page 40, library addresses are generally 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 Figure 16 on page 40. 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 18 on page 42).

Offset from Start of TCA (Register 12) Decimal	Offset from Start of TCA (Register 12) Hex	Name of Module Entry Point	Use
72	48	IBMBPGRD	Stack overflow routine to get VDA
84	54	IBMBEFL	FLOW module
108	6C	IBMBPGRA	Get non-LIFO dynamic storage
112	70	IBMBPGRB	Free non-LIFO dynamic storage
116	74	IBMBPGRC	Stack overflow routine for prolog
120	78	IBMBERRB	Error handler software interrupt
264	108	IBMBJWTA	WAIT module
268	10C	IBMBTOCA	Completion pseudo-variable routine
272	110	IBMBTOCB	Event variable assignment routine

Figure 18. Offsets Where Addresses of Library Modules Are Held in the TCA

DO-LOOPS

Where possible, DO-loops are carried out by means of a BXLE instruction, because this 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 a number of typical DO-loops is shown below. Note that the code will differ according to the content of the loop.

Source program

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

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.

IELCGIX   Stream I/O input—provides address of source of next edit-directed data or format item

IELCGIB   Stream I/O input—housekeeping after transmission of data item

IELCGOG   Stream I/O output—provides address of target of next edit-directed data or format item

IELCGOH   Stream I/O output—updates FCB, counts data item, and frees VDA if one was used

IELCGOC	Stream I/O—processes X format items
IELCGMY	Move long (registers 6,7,8,9)
IELCGCY	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 when they are used in a program.

OPTIMIZATION AND ITS EFFECTS

Optimization is the attempt to produce the most efficient possible object program. The OS 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 OS 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 show 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 cover all optimization carried out by the compiler.

Elimination of Common Expressions

Elimination of common expressions is handled by avoiding multiple calculations of the same expression, the value being retained 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
00005E  78 00 D 0BC              LE    0,B
000062  7A 00 D 0C0              AE    0,C
000066  70 00 D 0B8              STE   0,A


* STATEMENT NUMBER  3
00006A  78 60 D 0C4              LE    6,X
00006E  79 60 D 0C8              CE    6,Y
000072  47 B0 2 020              BNL   CL.2
000076  78 60 D 0C8              LE    6,Y
00007A  70 60 D 0C4              STE   6,X


* STATEMENT NUMBER  4
00007E                   CL.2    EQU   *

* CALCULATION OF COMMONED EXPRESSION FOLLOWS
00007E  70 00 D 0CC              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
00005E  78 00 D 0BC              LE    0,B
000062  7A 00 D 0C0              AE    0,C
000066  38 20                    LER   2,0
000068  70 20 D 0B8              STE   2,A


* STATEMENT NUMBER  3
00006C  78 60 D 0C4              LE    6,X
000070  79 60 D 0C8              CE    6,Y
000074  47 B0 2 022              BNL   CL.2
000078  78 20 3 01C              LE    2,28(0,3)
00007C  70 20 D 0B8              STE   2,A


* STATEMENT NUMBER  4
000080                   CL.2    EQU   *

* CALCULATION OF COMMONED EXPRESSION FOLLOWS
000080  70 00 D 0CC              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

```
2          Do I=1 TO N;
3          J=3;
4          END;
```

Object program

```
* STATEMENT NUMBER  2
00005E  48 E0 D 0BA              LH    14,N
000062  18 BE                    LR    11,14
000064  48 A0 3 018              LH    10,24(0,3)
000068  18 5A                    LR    5,10
00006A  40 50 D 0B8              STH   5,I
00006E  19 5B                    CR    5,11
000070  47 20 2 026              BH    CL.3
000074              CL.2         EQU   *


* STATEMENT NUMBER  3
000074  48 60 3 01A              LH    6,26(0,3)
000078  40 60 D 0BC              STH   6,J
```

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

```
* STATEMENT NUMBER  2

* STATEMENT LABEL          L
00005E  78 00 D 0B8              LE    0,X
000062  79 00 D 0BC              CE    0,Y
000066  47 20 2 038              BH    BED


* STATEMENT NUMBER  3
00006A  48 60 D 0C6              LH    6,I
00006E  4B 60 D 0C8              SH    6,N
000072  40 60 D 0C4              STH   6,J


* STATEMENT NUMBER  4
000076  50 60 D 0E0              ST    6,224(0,13)
00007A  48 60 3 01C              LH    6,28(0,3)
00007E  40 60 D 0E0              STH   6,224(0,13)
000082  97 80 D 0E2              XI    226(13),X'80'
000086  78 60 D 0E0              LE    6,224(0,13)
```

```
00008A   7B 60 3 01C                    SE    6,28(0,3)
00008E   3A 60                          AER   6,0
000090   70 60 D 0B8                    STE   6,X


*  STATEMENT NUMBER  5
000094   07 F2                          BR    2


*  STATEMENT NUMBER  6

*  STATEMENT LABEL          BED
000096   70 00 D 0C0                    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:

"6,6,6,7 STATEMENT MAY NEVER BE EXECUTED. STATEMENTS 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
00006A  7A 20 D 0A4       AE   2,B
00006E  70 20 D 0A0       STE  2,X
```

or

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

Object program

```
* STATEMENT NUMBER  6
0000E2  78 40 D 0BC       LE   4,B       Load B
0000E6  3C 44             MER  4,4       B**2
0000E8  38 64             LER  6,4
0000EA  3A 66             AER  6,6       2*B**2
0000EC  3A 64             AER  6,4       3*B**2
0000EE  70 60 D 0B8       STE  6,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 in Figure 19 on page 49, 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 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 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, 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.

```
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
00005E  48 60 3 018        LH    6,24(0,3)      Pick up 1 from static
000062  40 60 D 0B8        STH   6,I            Place in I

* INITIALIZATION CODE FOR OPTIMIZED LOOP FOLLOWS

* CODE MOVED FROM STATEMENT NUMBER 5
000066  48 E0 3 01A        LH    14,26(0,3)     Load 4 into R14 from static
00006A  48 80 3 01C        LH    8,28(0,3)      Load 40 into R8 from static
00006E  18 B8              LR    11,8           Load 40 into R11 for BXLE
000070  18 AE              LR    10,14          Load 4 into R10
000072  18 5E              LR    5,14           Load 4 into R5

* CONTINUATION OF STATEMENT NUMBER  4
000074               CL.2  EQU   *


* STATEMENT NUMBER  5
000074  78 05 D 0BC        LE    0,V0..B(5)     Pick up V0..B+R5
000078  70 05 D 0E4        STE   0,V0..C(5)     Place in V0..C+R5


* STATEMENT NUMBER  6
00007C  87 5A 2 016        BXLE  5,10,CL.2      Increment R5 by 4, test
                                                for end of loop, and
                                                branch or continue
```

Figure 19. Modification of DO-Loop Control Variable

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 20 on page 50, 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.

<u>Source program</u>

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

<u>Object program</u>

```
* 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 20.   Branching Around Redundant Expressions

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 of 768 will be picked up from the same address in both cases. Similarly, compiler-generated control descriptors (see Chapter 4, "Communication between Routines" on page 64, are generated only once if a number of variables require identical control information.

The process of avoiding duplication is known as <u>commoning</u>. It should be noted that constants may not be commoned if they are not used in the same way. In the example in Figure 21 on page 51, constant '123' is stored in a different form for assignment, multiplication, and exponentiation.

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

Object program

```
    00005E  78 00 3 01C         LE      0,28(0,3)   /*COMMONED ITEM*/
    000062  70 00 D 0B8         STE     0,X


    * STATEMENT NUMBER  3
    000066  78 20 D 0C0         LE      2,Z
    00006A  7C 20 3 01C         ME      2,28(0,3)
    00006E  70 20 D 0BC         STE     2,Y


    * STATEMENT NUMBER  4
    000072  41 70 D 0C4         LA      7,V
    000076  50 70 3 024         ST      7,36(0,3)
    00007A  50 70 3 02C         ST      7,44(0,3)
    00007E  96 80 3 02C         OI      44(3),X'80'
    000082  41 10 3 024         LA      1,36(0,3)
    000086  58 F0 3 014         L       15,A..IBMBMXSA
    00008A  05 EF               BALR    14,15


    * STATEMENT NUMBER  5
    00008C  78 00 3 01C         LE      0,28(0,3)
    000090  70 00 D 0C8         STE     0,A         /*COMMONED ITEM*/
```

Figure 21.   Use of Common Constants

The INTERRUPT Option

If the INTERRUPT option is in effect during compilation, extra
code is inserted in the compiled program to poll for attention
interrupts. The code is inserted at branch in points and before
END and RETURN statements of procedures. The code takes the
form:

```
    L    15, 288(12)
    BALR 14, 15
```

At 288 off register 12 is the TCA field TATP, which is the
address of a branch instruction. Normally the branch is simply
a return on register 14 to the polling point in compiled code.
However when the attention condition is raised the address is
altered to an entry point in IBMBEATA (IBMBEATB). Thus if an
attention interrupt has occurred the ATTENTION condition will be
raised after polling. The change of address in TATP is
specified by the STAX macro instruction issued during program
initialization if any of the procedures in the load module are
compiled with the INTERRUPT option.

FETCH AND RELEASE STATEMENTS

The PL/I FETCH and RELEASE statements are implemented by
compiled code calling the library module IBMBPFR, and by having
a PRV offset field for each module that is mentioned in a FETCH
statement.

For a FETCH statement, IBMBPFR checks in the PRV to see if the
module specified in the FETCH statement has already been loaded.
If it has been loaded, control is returned to the compiled code.
If it has not been loaded, IBMBPFR issues a LOAD macro
instruction and places the address of the loaded module in the
PRV. IBMBPFR then creates a FETCH control block (FECB) which it

attaches to the chain of FECBs addressed from the TIA. Control is then returned to compiled code.

For a RELEASE statement, IBMBPFR issues a DELETE macro instruction, and sets the address field in the PRV to the value of the PRV initialization word.

CHAPTER 3.   THE PL/I LIBRARIES

This chapter explains the use of libraries by the OS PL/I
Optimizing Compiler. The topics covered are: when and why
library routines are called, why there is both a transient
library 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.
Also covered are the multitasking and shared libraries.

The OS PL/I Optimizing Compiler is designed to be used in
conjunction with the OS PL/I Resident Library and the OS 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 OS PL/I subroutine library is divided into two separate
program products: the OS PL/I Resident Library (Program Number
5734-LM4) and the OS PL/I Transient Library (Program Number
5734-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, the
storage management routines 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 OS PL/I libraries reside on three direct-access data sets.
The resident library is on SYS1.PLIBASE and SYS1.PLITASK. The
transient library resides on SYS1.PLILINK.

The internal logic of individual library modules is described in
the publications OS PL/I Resident Library: Program Logic and OS
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 appears in the linkage editor map and the object-program
listing). The majority of the modules in the OS resident
library have a control name with the fourth letter <u>B</u> for example
IBMBOCL. This module has a link-edit name of IBMBOCLA. Some
modules, however, have a fourth letter T in their <u>control name</u>,
indicating that they are used only in a multitasking environment
and some a fourth letter F indicating that they are for use when
operating under CICS. The <u>link-edit</u> names of these modules
nevertheless have a fourth letter B. An example of this is the
multitasking priority-alteration routine IBMTTPRA. The
link-edit name for this module is IBM<u>B</u>TPRA. (See Figure 22 on
page 55.)

The result of this arrangement is that a number of library
modules can share the same link-edit name. Consequently, the
compiler can generate the same code regardless of whether the
program is going to operate in a multitasking, non-multitasking,
or CICS environment. Some CICS modules are held in
SYS1.PLIBASE, and others are link-edited during installation and
held as load module DFHSAP in SYS1.PLILINK.

Entry point names are given additional letters alphabetically.
The primary entry point name usually ends in an "A." Other
entry points are named "B", "C", etc. For example, the primary
entry point of the module with control name IBMBOCL is IBMBOCLA
and the secondary entry point is IBMBOCLB. Many modules have
only one entry point, such as the PL/I modules that work with
CICS, and are listed by the full entry point name. The module
names in SYS1.PLIBASE is almost always the primary entry point
name.

The naming convention for conversion modules is slightly
different. Arithmetic conversion modules have entry points
indicated by a two-letter mnemonic code.

THE MULTITASKING LIBRARY

The resident library is held on two data sets: SYS1.PLIBASE and
SYS1.PLITASK. SYS1.PLIBASE holds all modules that are needed to
execute non-multitasking programs. SYS1.PLITASK holds the
multitasking versions of all modules that differ for
multitasking and non-multitasking environments.

As explained above, both multitasking and non-multitasking
modules have the same link-edit names for their entry points.
Multitasking modules have a fourth letter T; non-multitasking
modules have a fourth letter B, in their control names.

The use of the same link-edit name permits the compiler to
generate the same code for library calls, regardless of whether
the program is multitasking or non-multitasking. For
multitasking programs, the data set SYS1.PLITASK must precede
SYS1.PLIBASE in input to the linkage editor. In this way, the
multitasking modules will be link-edited and the program will
run in a multitasking environment. Further details of this
arrangement are given in Chapter 14, "Multitasking" on page 307.

```
        ┌─────────────────────────────────────────────┬──────────┐
        │           CONTROL NAME                       │ EXAMPLES │
        │                                              ├──────────┤
        │         ┌─────┬─────┬─────┐                  │ IBMTPIRA │
        │         │     │  B  │     │                  │ IBMBEOCA │
        │         │ IBM │  T  │ xyz │                  │ IBMJWTA  │
        │         │     │  F  │     │                  │ IBMFEAIA │
        │         └─────┴─────┴─────┘                  │          │
        │                                              │          │
        │  ┌───────────┐ ┌──────────────┐ ┌──────────┐ │          │
        │  V           V                V              │          │
        │ ┌─────────┐ ┌──────────────┐ ┌──────────┐   │          │
        │ │Identify module│ │B=Base Module     │ │Mnemonic of│  │          │
        │ │as part of a   │ │T=Multitasking Module│ │module's │ │          │
        │ │PL/I Library   │ │F=CICS            │ │functions │   │          │
        ├─────────────────────────────────────────────┼──────────┤
        │           LINK-EDIT NAME                     │ IBMBPIRA │
        │                │                             │ IBMBEOCA │
        │                                              │ IBMJWTA  │
        │             IBMBxyz                          │          │
        ├─────────────────────────────────────────────┼──────────┤
        │           ENTRY POINT NAMES                  │          │
        │                │                             │          │
        │        ┌───────┴───────┐                     │          │
        │        │               │                     │          │
        │ Resident library modules   Transient library modules │          │
        │        │               │                     │          │
        │ Link-edit name followed by  Control name followed by │ IBMBPIRA │
        │     A,B,C, etc.              A,B,C, etc.      │ IBMBREFA │
        ├─────────────────────────────────────────────┼──────────┤
        │ Conversion modules sometimes have only two mnemonic │          │
        │ letters to identify the function,            │ IBMBCH   │
        │                                              │          │
        │ and use two mnemonic letters to identify entry points. │ IBMBCHXD │
        └─────────────────────────────────────────────┴──────────┘
```

Figure 22. Library Module Naming Conventions

LIBRARY WORKSPACE

DSAs (dynamic storage areas) for certain library routines 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 preformatted DSAs. These DSAs are known as levels of
library workspace. Their format can be seen in Figure 23 on
page 56. 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,
as the space is already allocated. The NAB pointer does not
have to be reset, because the next available byte is not
changed.

```
         >
    ┌──────┐  ┌─────────────────┬─────────────────────┐      ┐
    │      │  │ Flags           │ Offset to ONCA      │      │
    │      │  ├─────────────────┴─────────────────────┤      │
    │      │  │ Chain back field (to last DSA)        │      │
    │      │  ├───────────────────────────────────────┤      │
    │      │  │ Standard save area                    │      │
    │      │  │                                       │      │
    │   ┌──┼──│─Address of next library workspace     │      │ 1st level
    │   │  │  ├───────────────────────────────────────┤      │
    │   │  │  │ Current NAB                           │      │
    │   │  │  ├───────────────────────────────────────┤      │
    │   │  │  │ Workspace for library modules         │      │
    │   │  │  │                                       │      │
    │   └─>│  ├─────────────────┬─────────────────────┤      ┘
    │      │  │ Flags           │ Offset to ONCA      │      ┐
    │      │  ├─────────────────┴─────────────────────┤      │
    └──────┼──│─Chains back field                     │      │
           │  ├───────────────────────────────────────┤      │
           │  │ Standard save area                    │      │
           │  │                                       │      │
           │  ├───────────────────────────────────────┤      │
           │  │ Address of this level of LWS          │      │ 2nd level
           │  │ (used only when addressing the ONCA)  │      │
           │  ├───────────────────────────────────────┤      │
           │  │ Current NAB                           │      │
           │  ├───────────────────────────────────────┤      │
           │  │ Workspace for library modules         │      │
           │  │                                       │      │
           │  ├───────────────────────────────────────┤      ┘
           │  │ Current ONCA                          │
           │  │                                       │
           └──┴───────────────────────────────────────┘
```
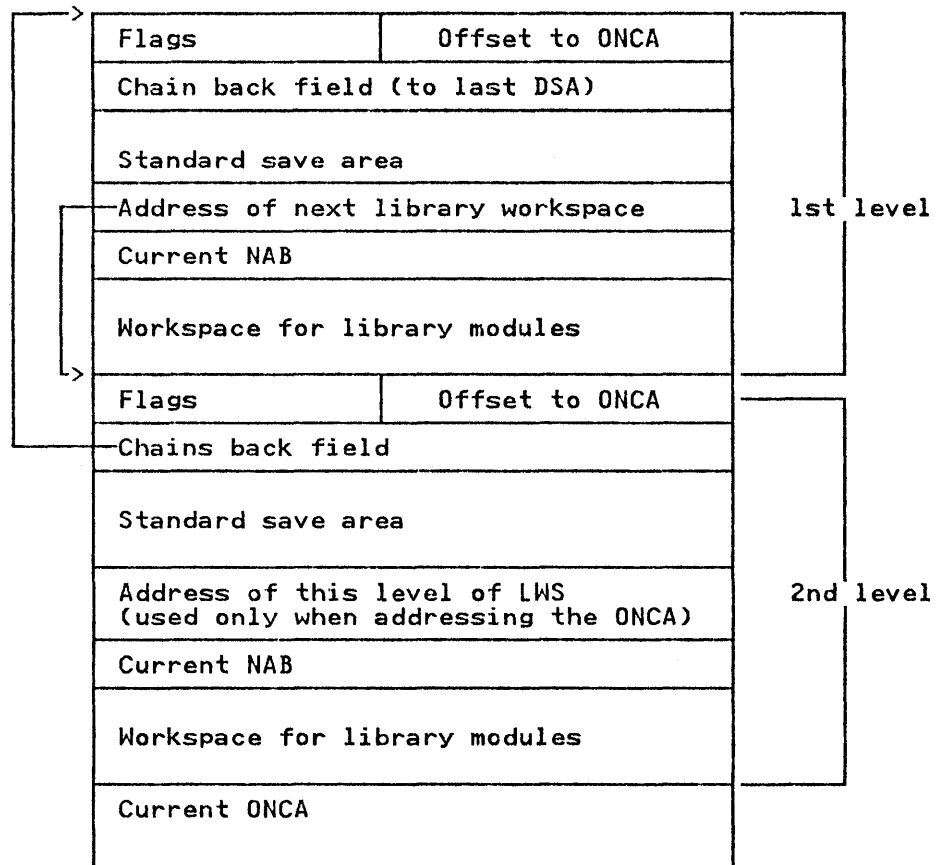
Figure 23.   Program Management Library Workspace

When it becomes possible that more than two LWS-type routines
might be active concurrently, two more library workspaces are
preallocated in LIFO space.

The library routines that can use library workspaces are those
that do not need working storage but can call other routines.
Library routines that need no working storage and do not call
other routines do not change R13.

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 23 illustrates the method of chaining
employed.

ALLOCATION OF LIBRARY WORKSPACE

Library workspace is originally allocated within the program
management area by the initialization routine IBMBPII. 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.
Library workspace is acquired by a subroutine of IBMBPIR that is
addressed from the TCA.

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 "Detecting the Occurrence of Conditions" on page 117.

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 so are 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 24. 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.
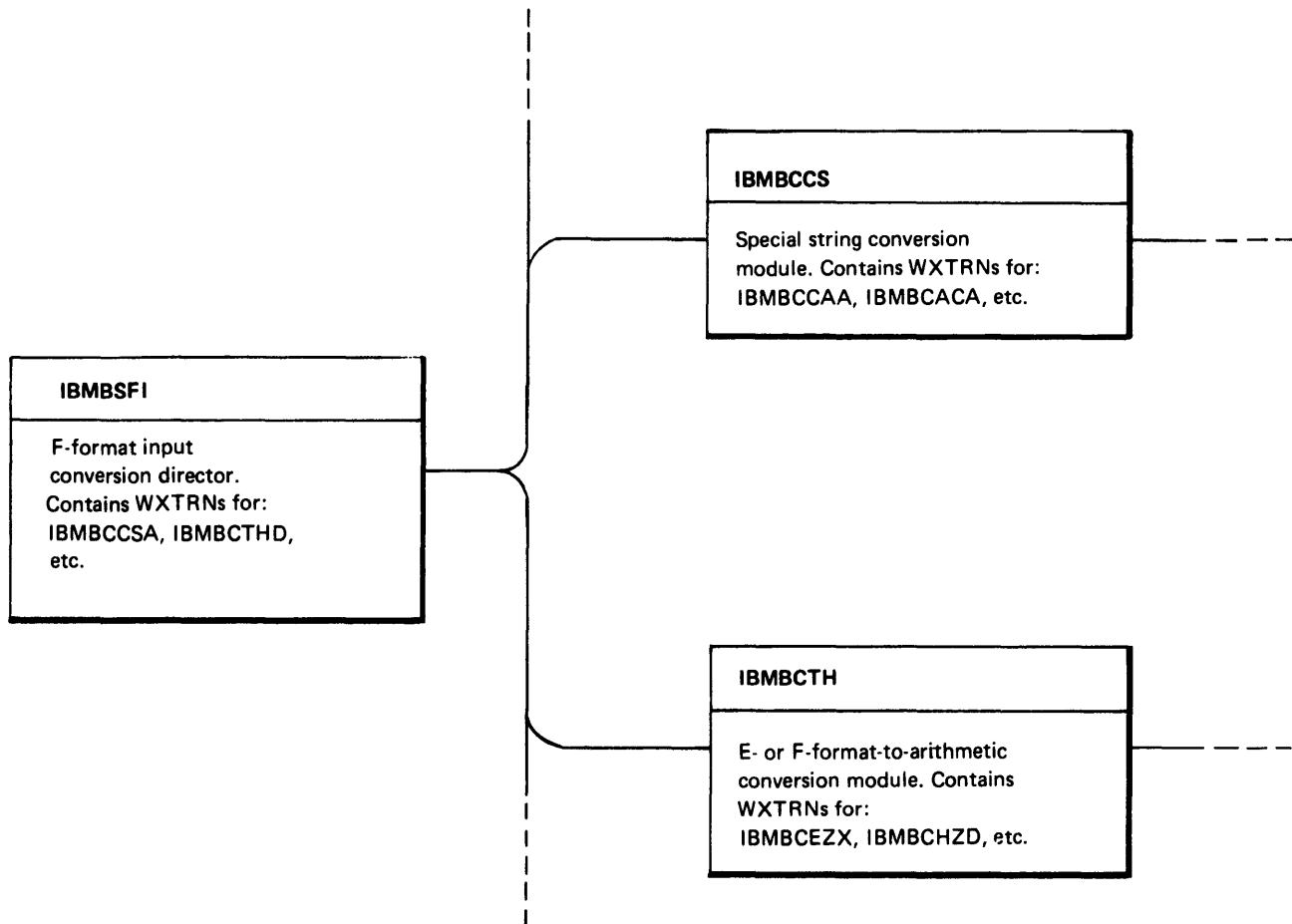
Figure 24. Example of Use of WXTRNs

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 24 on page 57 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 the required modules.

THE SHARED LIBRARY

The **shared library** is a PL/I facility that allows an MVS installation to load PL/I resident library modules into the link-pack area (LPA) so that they are available to all PL/I programs. This reduces space overheads.

The modules to be included in the shared library can be chosen by the installation. They must include the initialization routine, the error handling routine, the open file routine, and all modules addressed from the TCA that are not identical in multitasking and non-multitasking programs. Further details on the shared library and the optional modules are described in: OS PL/I Optimizing Compiler: Installation for MVS.

The routines in the shared library are held in two of three link-pack-area modules: IBMBPSM, and either IBMBPSL or its multitasking equivalent IBMTPSL. Each of the link-pack modules contains a number of library routines, and is headed by an addressing control block known as a **transfer vector**. IBMBPSM contains those modules in the shared library that are common to both multitasking and non-multitasking PL/I environments. IBMBPSL contains the non-multitasking versions of those modules that are not identical in multitasking and non-multitasking PL/I environments. This module has a multitasking counterpart, IBMTPSL, which holds the multitasking versions of such modules.

Two further modules are also involved in handling the shared library. These are the shared library addressing modules IBMBPSR and its multitasking counterpart IBMTPSR. One or other of these modules is link-edited with compiled code and held in the program region: IBMBPSR for non-multitasking programs, or IBMTPSR for multitasking programs. These two modules often use the alias PLISHRE. IBMBPSR and its multitasking counterpart hold dummy entry points, called stubs, that duplicate the names of all entry points of modules within the shared library. References to such entry points in compiled code are resolved to the stubs in IBMBPSR or IBMTPSR.

The situation during execution is shown in Figure 25 on page 59. In the link-pack-area are two link-pack modules: IBMBPSM and IBMBPSL (or its multitasking counterpart); these contain all the routines in the shared library. In the program region is the shared library addressing module IBMBPSR (or its multitasking counterpart). All references by compiled code to entry points in the shared library have been resolved by the linkage editor to IBMBPSR (or IBMTPSR).
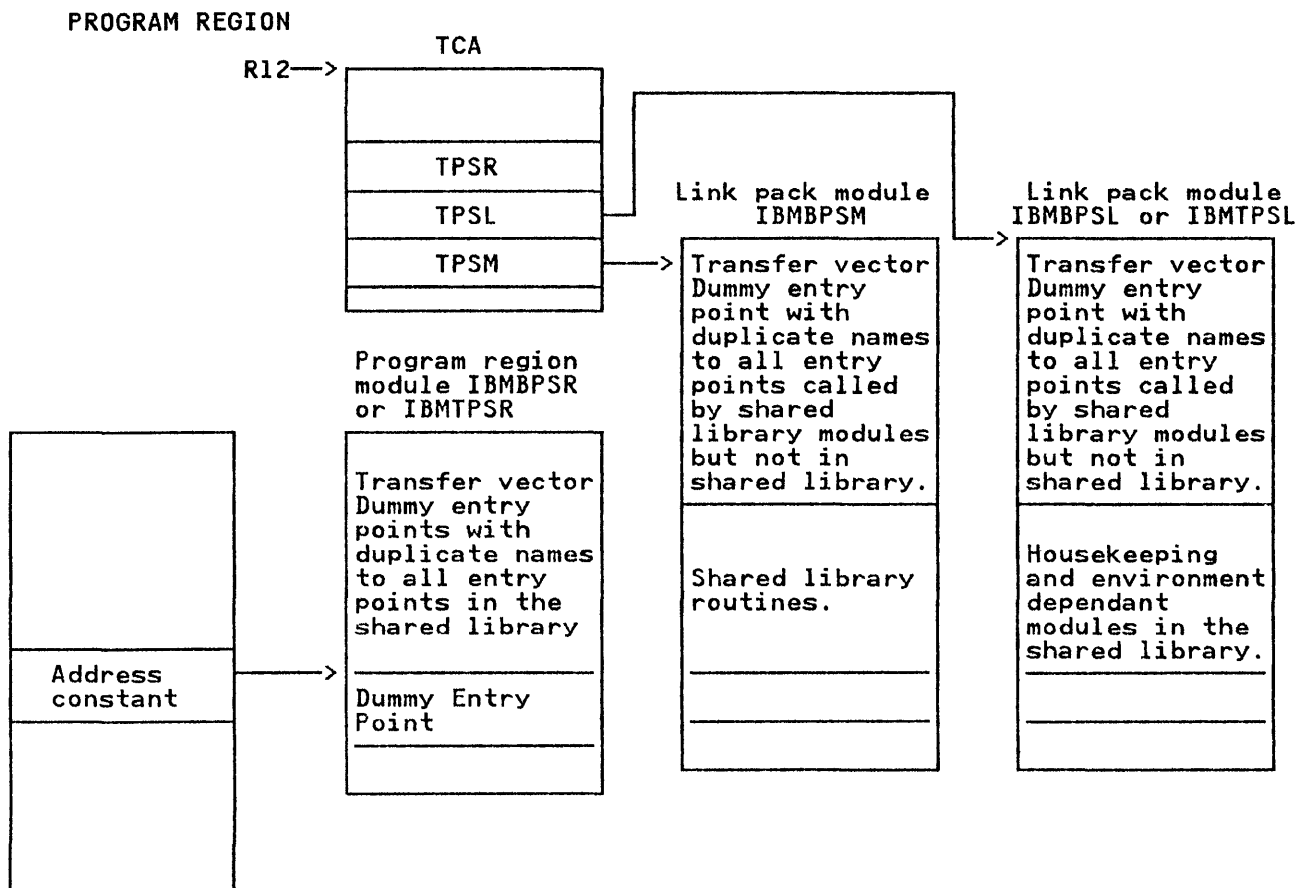
PROGRAM REGION

Figure 25.   The Shared Library During Execution

Communication between Program Region and Link-Pack-Area

Communication between the link-pack-area and the program region is handled by the transfer vectors that are held at the head of each module. Communication is necessary in both directions. The compiled program will need to call library subroutines that are held within the link-pack modules in the link-pack-area. Similarly, certain of the modules in the link-pack-area may need to call modules that are not included in the shared library. The link-pack-area modules IBMBPSL and IBMBPSM, are headed by transfer vectors, which are followed by the individual library modules in the shared library. The individual modules and the transfer vector are link-edited to form one module when the shared library is created.

The program region module IBMBPSR consists of a transfer vector and stubs. (The format of the shared library modules is shown in Figure 26 on page 60.) During program initialization, the addresses of the three modules being used (and consequently the address of the transfer vector) are placed in the TCA.

```
       Program region                    Link pack area (shared library
     IBMBPSR and IBMTPSR                   IBMBPSL, IBMBPSM & IBMTPSM
  ┌─────────────────────────┐       ┌  ┌─────────────────────────────┐
  │ VCONs and WXTRNs for     │       │  │ VCONs for all modules        │
  │ all modules that may     │       │  │ held in the shared           │
  │ be called from shared    │       │  │ library.                     │
  │ library but are not      │       │  ├─────────────────────────────┤
  │ included in it.          │       │  │ Dummy entry points all       │
  ├─────────────────────────┤  ─Transfer──  │ modules that may be          │
  │ Dummy entry points       │     vectors  │ called from the              │
  │ for all modules in       │       │  │ shared library but are       │
  │ the shared library,      │       │  │ held in the program          │
  │ followed by addressing   │       │  │ region. These entry          │
  │ code the passes          │       │  │ points are followed by       │
  │ control to the real      │       │  │ addressing code that         │
  │ modules.                 │       │  │ passes control to the        │
  └─────────────────────────┘       │  │ real modules.                │
                                     └  ├─────────────────────────────┤
                                        │ Individual modules           │
                                        │ in the shared                │
                                     Library  library.                      │
                                     routines ├────────────────────────┤
                                        │                              │
                                        ├──────────────────────────────┤
                                        │                              │
                                        ├──────────────────────────────┤
                                        │                              │
                                        └──────────────────────────────┘
```

Figure 26.   The Format of Shared Library Modules

The transfer vectors contain three types of data:

1. Dummy entry points for all modules that are not held in that
    area (that is, the program region transfer vector contains
    dummies for all entry points that are held in the shared
    library; the link-pack transfer vector contains entry points
    for all modules that could be called from the shared library
    but are not included in it).

2. Code, following the dummy entry points, that passes control
    from the dummy entry point in one area to the real entry
    point in another area.   The code takes the form:

    L   15,   offset(12)

    L   15,   xxx(15)

    BR   15

    where:

    offset
         is the address of IBMBPSM, IBMBPSL, or IBMPSR.

    xxx
         is the displacement into a table of V-type address
         constants (VCONs).

    The code (2) transfers control in the manner shown in
    Figure 27 on page 61.

    a. It picks up the address of the relevant transfer vector
        from the TCA, where it was placed during program
        initialization.

b. It picks up the address of the module it requires from a
    known offset from the start of the transfer vector.

c. It branches to the address, thus passing control to the
    required library routine.

The code does not use any register except register 15. The
link register (14) is not altered, and control returns
directly from the module to the caller.

3. An ordered list of addresses for all routines that are held
    in the same area as the vector. Addressing is shown in
    Figure 27.
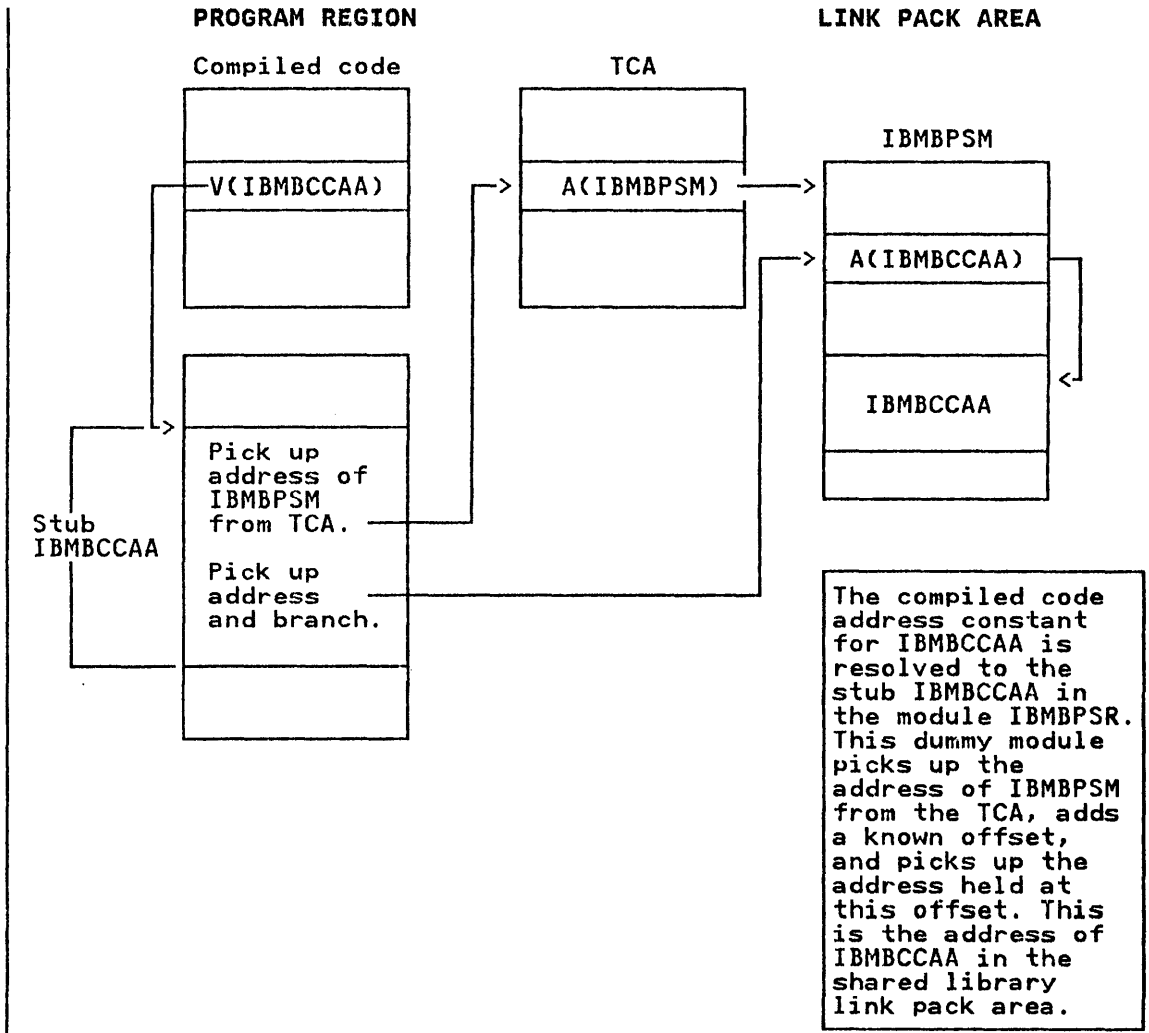
Figure 27.   Addressing a Module in the Shared Library

Execution When Using the Shared Library

Use of the shared library is specified by the linkage editor statement INCLUDE PLISHRE. PLISHRE is an alias for the program region modules IBMBPSR and IBMTPSR. The appropriate module will therefore be loaded by the linkage editor (IBMBPSR for non-multitasking programs; IBMTPSR for multitasking programs). All compiled code external references to shared library module entry points are then resolved to the dummy entry points in IBMBPSR (or IBMTPSR). Similarly WXTRNs in the program region module are resolved if compiled code issues an EXTRN for the entry point.

Program Initialization

At the start of the program, control is passed to one of the entry points of the initialization routine. This entry point will, in fact, be a dummy entry point in the shared library program region module. Each entry point is followed by code which requests the system to load the shared library link-pack modules. If the modules are already loaded, the system simply returns their addresses. If they are not loaded, it loads them into the link-pack-area, and then returns the addresses.

The addresses of the two link-pack-area modules and of IBMBPSR are added to the parameter list for IBMBPIR. IBMBPIR is then called in the usual shared library manner, that is via the transfer vector in one of the link-pack modules.

It is the standard action of the initialization routines to load these parameters into the appropriate fields in the TCA. When the shared library is not in use, meaningless information is loaded into these fields. However, as they are only accessed by the shared library modules, this does no harm.

Initializing the Shared Library

The shared library is initialized by the use of the PLISHR macro instructions, as described in OS PL/I Optimizing Compiler: Installation for MVS.

All five modules must be created at the same time. During the process, the table of VCONs in the link-pack modules, transfer vectors are generated, and the offsets to these VCONs from the head of the transfer vector are placed in the code following the dummy entry points in the program region modules.

A similar process is carried out for addresses in the program region. The VCONs within the link-pack modules are resolved by the linkage editor when the link-pack modules are created. The VCONs within the program region modules are qualified by WXTRNs, and are only resolved if compiled code generates an EXTRN for the entry point. Such EXTRNs are generated when required, as a normal part of the compilation process, regardless of whether the shared library is being used. The VCONs in the program region modules are resolved by the linkage editor when the program is link-edited.

For further details on the options available in the shared library, see OS PL/I Optimizing Compiler: Installation for MVS.

Multitasking Considerations

The shared library has been designed so that multitasking does not affect it. If PLITASK is specified before PLIBASE, the linkage editor statement INCLUDE PLISHRE will result in the module IBMTPSR being loaded and linked in the program region. When control passes to the code following the IBMBPIR entry point in IBMTPSR, a request is made to the system to load the multitasking shared library module IBMTPSM. The program then runs in the usual manner, with the multitasking modules.

An installation can specify a shared library that includes only
the multitasking or the non-multitasking modules. However, both
multitasking and non-multitasking versions of the program region
module will still be created. The module for the unwanted
environment will be a dummy. This prevents problems should an
INCLUDE PLISHRE statement be included in a program that is
intended to run in the environment with no shared library. If
this process was not carried out, such a statement could result
in the incorrect environment being initialized.

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, adjustable bounds or extents may prevent the data attributes being fully known at compile time, or the data may be being passed to another PL/I procedure or 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 control blocks are:

DESCRIPTORS: These hold the extent of the data item (that is, string lengths, array bounds, and area sizes).

LOCATORS: These hold the address of a data item and, if they are not concatenated with the descriptor, hold the descriptors address.

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.

DESCRIPTOR/LOCATOR: This is a term used to describe the control block consisting of a descriptor concatenated with a locator.

An example of the way in which data is related to its locators, descriptors, and DEDs is given in Figure 28 on page 65.

```
                              Storage
                        ┌───────────────────────┐
PL/I Statement      ┌──>│ TABLE (0)             │  <────Virtual origin
DCL TABLE(10)       │   ├───────────────────────┤
FLOAT DECIMAL (6);  │   │ TABLE (1)             │
                    │   └───────────────────────┘
                    │   ┌───────────────────────┐
                    │   │ TABLE (9)             │      Array TABLE (10)
                    │   ├───────────────────────┤
                    │   │ TABLE (10)            │
                    │   └───────────────────────┘
                    │   ┌───────────────────────┐
                    │   │ Short floating-point decimal 6│  DED
                    │   │                       │
                    │   └───────────────────────┘
                    │   ┌───────────────────────┐
                    └───│─Address of TABLE      │      Aggregate locator
                        ├───────────────────────┤
                    ┌───│─Address of descriptor │
                    │   └───────────────────────┘
                    │   ┌───────────────────────┐
                    └──>│       *RVO=4          │
                        ├───────────────────────┤
                        │ Multiplier=4          │      Array descriptor
                        ├───────────┬───────────┤
                        │Upperbound=10│Lowerbound=1│
                        └───────────┴───────────┘
```
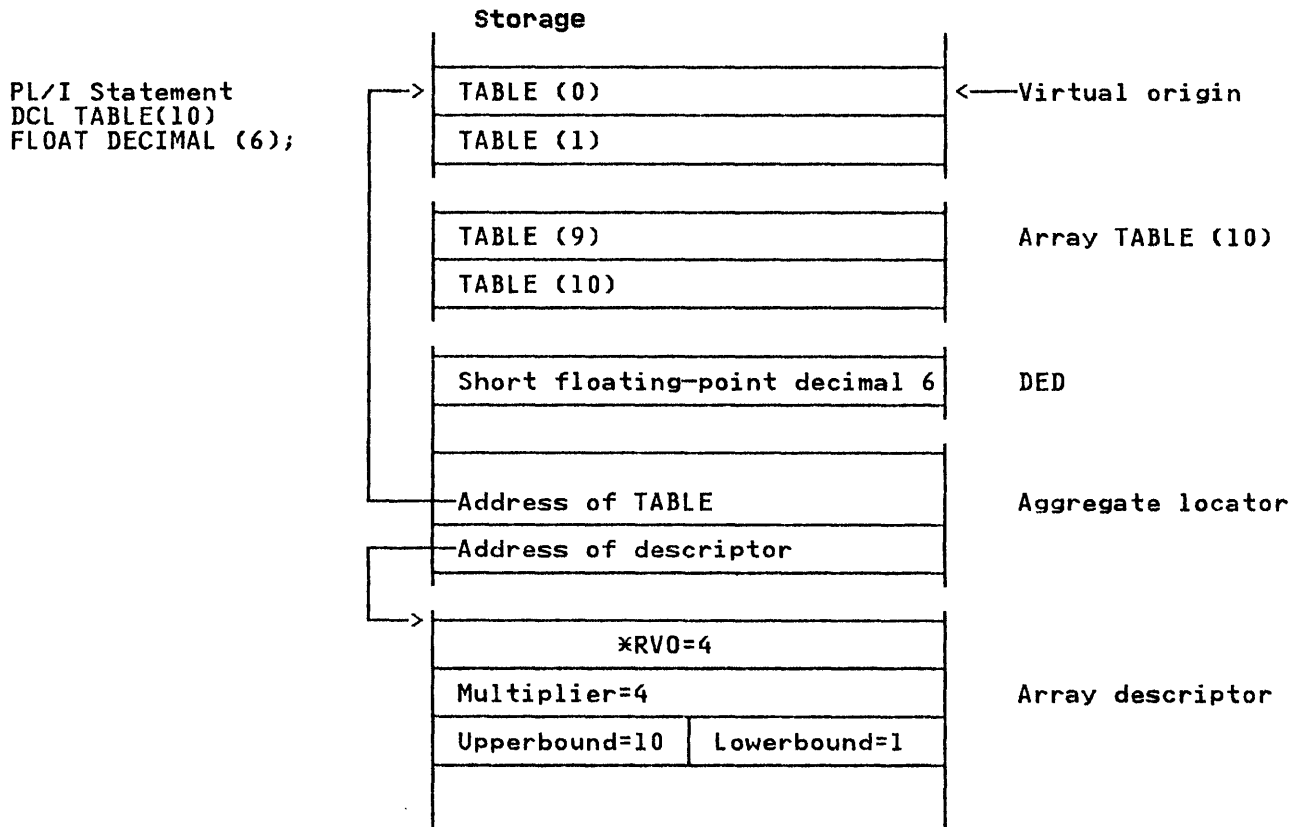
*RVO (Relative virtual origin) is
the offset of the actual origin
of the array from the virtual
origin (the position that element
TABLE (0) would hold if it existed)

Figure 28.   Example of Descriptor, Locator, DED, and Storage Location of an Array

Passing Arguments and Returned Values

When arguments are passed between PL/I routines register 1 is
used to point to a list of addresses known as a parameter list.
These addresses are the addresses of the data items for
nonaggregate arithmetic items. For other items, where the
receiving routines may be expecting data about length or format
in addition to the data itself, descriptors and locators are
used. For control information, such as files or entries other
control blocks are used because the item itself cannot properly
be said to have an address. The addresses within a parameter
list are shown below.

Data Type Passed	Address Passed
Arithmetic items	Data
Array or structure	Aggregate locator
String or area	Locator/descriptor
File constant (passed to resident library subroutines only)	DCLCB

Data Type Passed	Address Passed
File constant or variable (passed to routines other than above)	File variable
Entry	Entry data control block
Label	Label data control block
Pointer	Data
Offset	Data
Task	Data (task variable)
Event	Data (event variable)

Locators and descriptors are described later in this chapter. A file variable is a full word holding the address of the DCLCB. The layout of all other control blocks is shown in Appendix A, "Control Blocks" on page 326.

The last entry in the list is marked by having the first bit in the word set to 1.

If a function reference is used the last field is used for the address of the returned value or its appropriate control block. Thus there is one more field in the parameter list than there are arguments passed.

PL/I only returns a return code when it returns to an outside caller via its termination routine. No return code is passed between PL/I procedures. When a non-PL/I routine that returns a return code is called, the value of the return code can be accessed if the procedure is declared with the RETCODE option. For example:

    DCL ASMSUB OPTIONS (ASSEMBLER,RETCODE);

For such entries the compiler generates code that saves the value returned in register 15 from such a program and makes it available when the PLIRETV built-in function is specified. The method of setting up the return code when PL/I returns to the system or an outside caller is described in " The Process of Termination." on page 80.

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 address of the first item in the array or structure.
Relative virtual origin (RVO)	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 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 prolog 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 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.

Except for controlled variables, descriptors and locators are always held in the static internal control section, regardless of the attributes of the data that they describe. Reentrant programs that require update are copied into AUTOMATIC storage.

For controlled variables, the descriptor and, sometimes, the locator are held immediately before the data. (For details see "Controlled Variable Block" on page 335).

The following types of descriptor and locator are generated. Figure 29 on page 68 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.

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 29. Descriptors, Locators, and Symbol Tables: When Generated, Where Held

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. For further details, see "String Locator/Descriptor" on page 402.

Area Locator/Descriptor

The area locator/descriptor holds the address of the start of the area and the length of the area, as shown in "Area Locator/Descriptor" on page 326.

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. This locator is shown in "Aggregate Locator" on page 330.

Array Descriptor

The array descriptor holds:

1. The relative virtual origin (RVO) of the array.

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

3. The multiplier for each dimension.

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 "Array Descriptor" on page 331.)

Structure Descriptor

The structure descriptor 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 structure descriptor descriptor. (See "Structure Descriptor" on page 403).

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 number of dimensions and logical level of all the structure elements, and the number of dimensions, 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 "Aggregate Descriptor Descriptor" on page 328. An example showing the method used to map a structure that contains an element with an adjustable extent is shown in Figure 30 on page 70.

Where possible, aggregate descriptor descriptors are commoned.

Declaration
DCL 1 A,
　　2 B FLOAT
　　2 C,
　　　3 D CHAR(N),
　　　3 E FLOAT;

DURING COMPILATION

1 Space for structure descriptor allocated in static storage.

SD　　　　　　　1

Space for offset of B
Space for offset of D
Space for descriptor of D
Space for offset of E
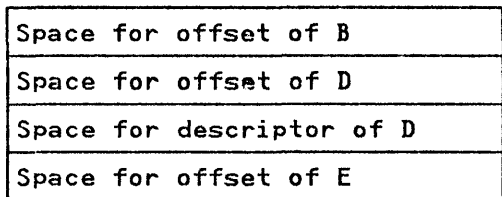
2 Aggregate descriptor descriptor allocated, and fields filled in from structure declaration.

ADD　　　　　　2

01	All ones		level 1	00	Zero
10	X'31'	X'4'	level 2	00	Zero
00	Zero		level 2	00	Zero
10	X'7'	Zero	level 3	00	Zero
11	X'31'	X'4'	level 3	00	Zero

3 Aggregate locator allocated, and address of structure descriptor place in second word.
Code is generated within the prolog 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.

AL　　　　　　　3

Space for address of structure
Address of structure descriptor
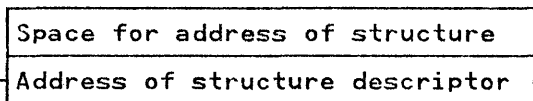
DURING EXECUTION

4 Prologue code places value 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.

SD　　　　　　　5

Zero		
X'7'		
4　1 byte	0	unused
X'8'		
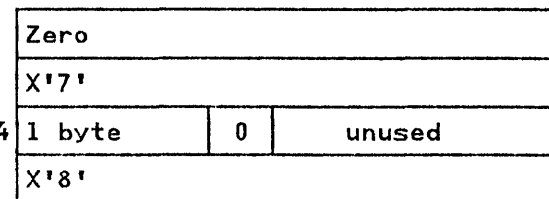
6 IBMBAMM returns the length of the structure to compiled code, which acquires a VDA for the structure and places the address of the structure in the aggregate locator.

ADD

ADD is unchanged during execution.

AL　　　　　　　6

Address of structure
Address of structure descriptor
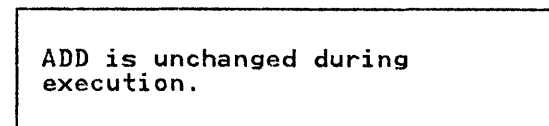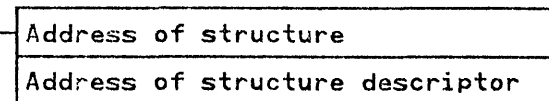
VDA for structure　6

B	
padding	D
E	
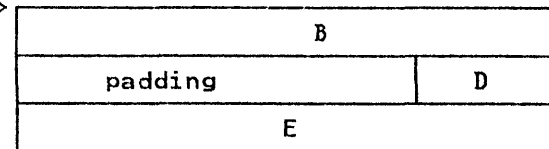
THE RESULT

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 30.　Example of Handling a Structure Containing an Adjustable Extent

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 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),                    DCL 1 ST,
      2 B,                             2 B(10),
      2 C;                             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 Appendix A, "Control Blocks" on page 326). 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 depends on the data type of the item. DEDs are shown in detail in "Data Element Descriptor (DED)" on page 337.

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: 4 bytes long.

ARITHMETIC PICTURED DEDS: (always decimal) 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: 4 bytes long.

PICTURED STRING DEDS: (always character string) 6 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 format of DEDs is shown under "Data Element Descriptor (DED)" on page 337.

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. Symbol tables hold the name of the variable, its address, and the address of its DED plus certain other information (see Appendix A).

GET DATA and PUT 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. The necessary 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 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 marks the end of the vector.

Figure 31 on page 73 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.

Get Data (A,B,C), Put 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 A. 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 X. Thus the control section for the external variable B

would be BX. Such a control section would also contain the DED
of the variable (or DEDs if the variable was a structure).

PROGRAM BLOCK STRUCTURE

Main procedure

```
    DCL A,B,C;
        Subroutine 1
        DCL X,Y,A;

        Subroutine 2
        DCL X,Y;
```

Symbol table vector   Symbol tables for:

Vector for
main procedure

Symbol table vector		Symbol tables for:
A	——————>	A in main procedure
B	——————>	B in main procedure
C	——————>	C in main procedure
00000........0		
00000........0		
X	——————>	X in subroutine 1
Y	——————>	Y in subroutine 1
A	——————>	A in subroutine 1
00000........0		
—Pointer		
X	——————>	X in subroutine 2
Y	——————>	Y in subroutine 2
00000........0		
—Pointer		

Vector for
subroutine 1

Vector for
subroutine 2

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

Figure 31. Symbol Tables and Symbol Table Vectors

CHAPTER 5.   OBJECT PROGRAM INITIALIZATION

Before the output from the compiler can be executed, 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. Initialization for
multitasking programs is explained in Chapter 14, "Multitasking"
on page 307. The chapter also gives a brief description of the
program management area; a control area set up during program
initialization.

LINK-EDITING

The functions and use of the linkage editor program are
described in the operating system publications. 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. A major module used in program
management is the error-handling module, IBMBERR. An external
reference to this module is contained in the PL/I initialization
routine, IBMBPIR. An external reference to IBMBPIR 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. This feature is also used
to handle data declared EXTERNAL. Control sections for each
such data item as STATIC EXTERNAL are generated by all programs
in which the data is declared. Only one of these is resolved.

Note: With the exception of interlanguage communication calls,
the entry statement cannot be used to pass control to a
specified PL/I program; entry must be made through the PLISTART
CSECT.   The PLISTART CSECT has three entry points:

        PLISTART
        PLICALLA
        PLICALLB.

These entry points are described in this chapter, and in the
"Communicating between PL/I and Assembler Language Modules"
chapter, of OS PL/I Optimizing Compiler: Programmer's Guide.
How PL/I handles entry into into another language is discussed
in Chapter 13, "Interlanguage Communication" on page 281.

The PLIMAIN control section is not generated by the compiler if
the PL/I source program does not contain the MAIN option.
However, a control section named PLIMAIN is included in the
module IBMBEMN. This control section contains the address of
code that calls the module IBMBPEP, which puts out a message
saying there is no main procedure, after which the program is
terminated.

PROGRAM INITIALIZATION

Code is compiled by the PL/I Optimizing Compiler on the assumption that various control blocks will have been set up and 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. An area for the allocation of PL/I dynamic storage should be available. This area is known as the initial storage area (ISA).

2. During initialization, a dynamic storage area (DSA) should exist. This will give the address of the start of the area available for dynamic storage allocation and will act as a save area for the calling routine's registers.

3. 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.

4. Program checks should be passed to the PL/I error-handling module IBMBERR.

5. Preformatted DSAs should exist for certain library routines. These preformatted DSAs are known as library workspace (LWS).

6. 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 ONCA.

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

The resident program initialization routine IBMBPIR, calls IBMBPII to acquire the ISA, and set up the various control blocks. These control blocks are in the head of the ISA known as the program management area. The contents of the program management area are described later in this chapter.

The default ISA size and other options are controlled either by the system default module IBMBOPT or by specifying an external variable called PLIXOPT within the program.

The use of initialization routines obviates the need for special code in main procedures, and allows two procedures with the MAIN option to be used in the same program.

As shown in Figure 32 on page 76, the initialization routine IBMBPIR is reentered after the execution of compiled code. This is done by the standard action of the epilog code. The registers of IBMBPIR are stored in the dummy DSA by the prolog code, and restored by the epilog code. When terminating the program, IBMBPIR calls IBMBPIT, to handle the majority of the termination functions.

```
┌─────────────────────────────────┐          ┌─────────────────────────────────┐
│  PLISTART                       │          │  Initialization routines        │
├─────────────────────────────────┤          ├─────────────────────────────────┤
│                                 │          │                                 │
│  Receives control from          │          │  Set up TCA, initialize storage │
│  system                         │          │  and issue SPIE/ESPIE and       │
│                                 │   ──>    │  STAE/ESTAE to initialize PL/I  │──┐
│                                 │          │  error-handling scheme.  Pass   │  │
│                                 │          │  control to the address in      │  │
│                                 │          │  PLIMAIN                        │  │
│                                 │          │                                 │  V
└─────────────────────────────────┘          └─────────────────────────────────┘
  ┌──────────────────────────────────<──────────────────────────────────────────┘
  │
  V ┌─────────────────────────────────┐          ┌─────────────────────────────────┐
    │  Prologue code                  │          │  Functional code                │
    ├─────────────────────────────────┤          ├─────────────────────────────────┤
    │                                 │          │                                 │
    │  Acquires DSA for main          │          │  Carries out function required  │
 ──>│  procedure, initializes         │   ──>    │  in source program. This        │──┐
    │  control blocks, etc.           │          │  usually involves calls to      │  │
    │                                 │          │  library subroutines.           │  │
    │                                 │          │                                 │  V
    └─────────────────────────────────┘          └─────────────────────────────────┘
  ┌──────────────────────────────────<──────────────────────────────────────────┘
  │
  V ┌─────────────────────────────────┐          ┌─────────────────────────────────┐
    │  Epilogue code                  │          │  Termination routines           │
    ├─────────────────────────────────┤          ├─────────────────────────────────┤
    │                                 │          │                                 │
    │  Restores IBMBPIR's             │   ──>    │  Closes any files still open and│
 ──>│  registers                      │          │  returns control to system.     │
    │                                 │          │                                 │
    │                                 │          │                                 │
    └─────────────────────────────────┘          └─────────────────────────────────┘
```

Figure 32.   Flow of Control During Execution

Fast-Path Initialization/Termination

For fast-path initialization/termination IBMBPII, IBMBPIT, and
the storage management routine IBMBPGR are linked together
during compiler installation and all loaded together with
IBMBPIR.

INITIALIZATION AND TERMINATION ROUTINES

Three routines are used in initialization and termination.   They
are:

IBMBPIR   Initialization/termination routine.

IBMBPII   Initialization routine.

IBMBPIT   Termination routine.

The resident routine, IBMBPIR, is a short control routine.   The
major functions are carried out by the transient routines.
However, IBMBPIR contains a number of housekeeping subroutines,
including code to handle GOTO out of block in certain abnormal
situations, and the STAE/ESTAE exit subroutine.   These are
described in Chapter 6, "Storage Management" on page 84, and
Chapter 7, "Error and Condition Handling" on page 105,
respectively.

IBMBPIR has three entry points. One of these is for use by the supervisor; the other two are for use by problem programs written in languages other than PL/I. The main difference between the entry points is the parameters that are expected. The entry points are:

IBMBPIRA   Used when entry is made from the system.

IBMBPIRB   For use by non-PL/I callers who wish to accept PL/I default ISA size.

IBMBPIRC   For use by non-PL/I callers to nominate the ISA and heap areas, or to override the other execution time options.

Entry points B and C will be used by programmers specifying PLICALLA and PLICALLB respectively. Using PLICALLA results in control being passed to IBMBPIRB. Using PLICALLB results in control being passed to IBMBPIRC.

IBMBPIRA and IBMBPIRC can be passed a number of parameters related to program management. These include ISASIZE, HEAP, and REPORT. Module IBMBPIR assumes that all parameters preceding a slash(/) are program management parameters. All main procedure parameters must, therefore, be preceded by a slash, otherwise they are taken to be parameters for PL/I program management.

Entry point IBMBPIRC can be passed a parameter list that contains the length and, optionally, the address at which the ISA is to begin. The ISA size and address are passed to IBMBPII. The format of the parameter list is described in "Communicating between PL/I and Assembler-Language Modules" chapter, of the OS PL/I Optimizing Compiler: Programmer's Guide.

Entry point IBMBPIRB cannot accept any program management parameters; the default ISA size is always given. (See also, "Acquiring the ISA" on page 78.)

The Process of Initialization

When IBMBPIR is called to initialize the program, it acquires workspace and calls IBMBPII. IBMBPII carries out the actions described below. An area large enough for both workspace and the ISA is acquired when the default is used, or when these conditions are all met:

*   The ISASIZE is positive

*   The ISASIZE is a reasonable size (4K to 16M bytes)

*   The ISA is supplied during compilation

Handling Execution-Time Options

IBMBPII, which contains the default execution time options, analyzes the options specified. These options, which were also known as program management parameters, can be specified in the following ways:

1. As parameters of the EXEC statement,

2. From an external variable called PLIXOPT in the PL/I program.

3. From the default module IBMBXOPT, which is usually set during installation. See the OS PL/I Installation Guide for your operating system.

All three sources may exist, and the options are merged from
them. IBMBPII first uses the default module IBMBXOPT. It then
searches for a control section called IBMBPOPT which is produced
by the compiler if an external variable called PLIXOPT is
declared in the program. Prior to release 3 of the optimizing
compiler, when IBMBPOPT did not exist, IBMBPII would search for
PLIXOPT which was left uncompiled by the compiler. Any options
specified in PLIXOPT are then merged with those in IBMBXOPT,
with the values in PLIXOPT overriding those in IBMBXOPT. The
process is then repeated with any execution time options
specified as parameters in the EXEC statement. When the
execution time options have been sorted out, IBMBPII carries out
the actions described below.

Acquiring the ISA

The method of acquiring storage for the ISA depends on the entry
point in IBMBPIR used.

If entry point C is used, and both the ISA size and address have
been passed, no further action need be taken.

If the ISA size has been passed, to either entry point C or
entry point A, a GETMAIN macro is issued for the amount of
storage requested.

If no ISA size has been specified, the default action is taken.
The default action is to obtain all the available storage. The
high-address half of this storage is then freed, and the lower
half retained as the ISA. If the resulting figure is not large
enough to hold the program management area an area large enough
for the program management area is obtained.

If there is not enough space for the ISA size requested, or if
the defaults do not provide enough space for the program
management area, the action described below under "Error
Situations" on page 80 is taken.

Initialization of the Program Management Area

The program management area is set up at the low address end of
the ISA. IBMBPII initializes the various control blocks. These
are shown in Figure 33 on page 79. Their functions are
described below under "The Program Management Area" on page 81.

The storage management routine is loaded, and the addresses of
its various entry points are placed in the TCA. If a storage
report is requested, module IBMBPGD is loaded; otherwise, module
IBMBPGR is used.

Initializing PL/I Error Handling

The PL/I error handling scheme handles all program checks, and
attempts to handle ABENDs. The address of the old fake PICA is
saved in the TCA so that the previous SPIE/ESPIE may be restored
during program termination, and SPIE/ESPIE and STAE/ESTAE macro
instructions are issued to set up the PL/I error handling
scheme.

The SPIE/ESPIE macro specifies entry into entry point A of the
error handling module IBMBERR. (This subroutine loads the ABEND
analyzing module IBMBPES.) A full description of the PL/I error
handling facilities is given in Chapter 7, "Error and Condition
Handling" on page 105.

When the program management area has been initialized, and the
SPIE/ESPIE and STAE/ESTAE macro instructions have been issued,
IBMBPII returns control to IBMBPIR.

IBMBPIR checks that the return has been normal and, if so,
points register 1 at the parameters for the main procedure, and
calls the procedure whose address is held in the control section
PLIMAIN.

R12——>

TCA Task communications area. See text and Appendix A
TCA Appendage See text and Appendix A
Dummy ONCA (ON communications area) Holds default values for condition built-in functions
TRT Table Translate-and-test table for IBMBERR, used in error handling to test for relevant on-cells.
Diagnostic File Block Contains information relating to the use of SYSPRINT for the transmission of diagnostic messages
Dump Block (DUB) Block used to access the dump file
Ordered delete list Used to hold a list of transient modules to be deleted during program termination
Dummy task variable Used in tasking if no task variable is declared
Save area for IBMBPGR Used by storage management routines when new segment of storage is required.
Dummy DSA (Dynamic storage area) (See Appendix A) Contains DSA for initialization routine, back-chain to calling routine's save area (if any), pointer to start of major free area (NAB), etc.
LWS (library workspace) Two preformatted DSAs for use by certain library routines
ONCA Space in which condition built-in function values are placed after an interrupt. Back-chain points to dummy ONCA
Pseudo Register Vector Control block used in addressing files and controlled variables.
Hold area for parameters passed to a main procedure.
MAJOR FREE AREA 4K to 16M Bytes

R13——> (aligned with Dummy DSA row)

Dummy
DSA
NAB——>

Figure 33.   Program Management Area

NOSPIE AND NOSTAE OPTIONS: If NOSPIE is specified in the parameters passed to IBMBPIR no SPIE/ESPIE macro is issued by the initialization routine. This allows an installation to specify its own method of handling program check interrupts. Similarly if NOSTAE is specified a STAE/ESTAE macro will not be issued.

If the INTERRUPT option was in effect during the compilation of any of the procedures in the load module, IBMBEAT is included in the load module. If it has been, a STAX macro instruction is issued so that attention interrupts will pass control to compiled code. For further information on interrupts, see "The INTERRUPT Option" on page 51.

The STAX exit, set up by the STAX macro instruction changes the contents of the TATP field in TCA. This field contains a pointer to an instruction that is executed by polling code in the compiled code or library modules at various convenient points in execution. Normally the instruction is a branch on register 14 resulting in the continuation of the program. However the STAX exit sets it to a branch into the error handler (IBMBERR) which then raises the ATTENTION condition.

Error Situations

If there is insufficient storage available to meet the requested ISA size, IBMBPII calls IBMBPEP, which puts out an "INSUFFICIENT MAIN STORAGE" message. IBMBPII then returns to IBMBPIR, requesting it to free the storage acquired, and terminate the program.

If no PL/I main procedure is provided, and there is no alternative PLIMAIN control section provided by the user, an error module is called. A "NO MAIN PROCEDURE" message is generated, and the program is terminated.

The Process of Termination.

When the main procedure is complete, epilog code for the main procedure returns control to IBMBPIR, passing to it the address of the DSA. If the termination is normal, IBMBPIR restores the value of register 13 to that passed to it in register 0, which is the main procedure DSA. IBMBPIR then sets flags in the TCA indicating that the program is terminating, and tests the THFN flag. If it is set on, this means that a FINISH ON-unit was activated and calls the error handler to raise the FINISH condition. If there is no GOTO from the FINISH ON-unit, the error handler returns to IBMBPIR using the GOTO-out-of-block mechanism. The flags set in the TCA to indicate program termination are tested and, as they are set, control is returned from the GOTO code in the TCA to the abnormal-GOTO subroutine in IBMBPIR.

The GOTO-out-of-block routine handles any outstanding housekeeping problems. Exit DSAs are correctly terminated. (A full discussion of the GOTO-out-of-block mechanism and its implications is given in "GOTO Statements" on page 36).

When IBMBPIR is entered again, count and flow information is printed or displayed. All files are then closed by calling IBMBOCL. IBMBPIR then calls IBMBPIT to complete the housekeeping.

IBMBPIT issues STAE/ESTAE and SPIE/ESPIE macro instructions to restore the error handling situation. Fetched procedures are then released and heap storage is freed. Diagnostic files are closed. In addition, the storage report is made if it is required.

IBMBPIT then checks the value in TCA field TORC to see if the ERROR condition was raised. When the ERROR condition is found in TORC, IBMBPIT passes control to IBMBEER. IBMBPIT passes to

IBMBEER the error code as a positive number if there was an ON
ERROR block called, or a negative number otherwise. The code
that IBMBEER returns to IBMBPIT is, in turn, passed to IBMBPIR.

If the ERROR condition is not found in the TCA field TORC,
control returns to IBMBPIR with a zero return code.

IBMBPIR checks the return code from IBMBPIT and places it in
register 15. If the code is zero, IBMBPIR frees the ISA and
either returns to the operating system or returns to the caller.
If the code is not zero, it ABENDs with the return code from
IBMBEER.

THE PROGRAM MANAGEMENT AREA

A diagram of much of the program management area is shown in
Figure 33 on page 79. It shows the situation when the compiled
program is called.

Translate-and-Test Table

The translate-and-test table contains code used in error
handling to identify relevant on-cells. (See Chapter 7, "Error
and Condition Handling" on page 105.)

Dump File Block

This is space used during the execution of PLIDUMP to hold the
DCB and other information for the dump file.

Loaded Module or Ordered Delete List

This is a list of modules that are deleted by IBMBPIR during
program termination. Certain transient modules that are not
deleted by other methods place their name in this list to ensure
that they are deleted when the program is terminated.

Dummy Tasks and Event Variables

These are included in the program management areas to allow the
use of the STATUS and PRIORITY built-in functions in
non-multitasking programs, and to allow multitasking programs to
operate if no task or event variables are explicitly declared.

Dummy DSA

The dummy DSA acts as a save area for the registers of the
initialization routine IBMBPIR, 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. This process
is described under "Searching for Established ON-Units" on
page 134. 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 preformatted DSAs that are used by certain
of the library modules. (See Chapter 3, "The PL/I Libraries" on
page 53.)

Pseudo-Register Vector

This is used in addressing files, fetched procedures, and controlled variables. (See Chapter 2, "Compiler Output" on page 12.)

MULTITASKING

The program initialization process for a multitasking environment is described in Chapter 14, "Multitasking" on page 307.

PROGRAM MANAGEMENT UNDER CICS

When operating under CICS, the optimizing compiler makes use of a slightly different program management scheme.

This is achieved by using special CICS-only library modules and the use of the CICS bootstrap module DFHPL1OI which acts as the entry point for all PL/I programs under CICS in a similar way that PLISTART is used under OS.

The CICS-only modules are loaded in place of the standard modules because an INCLUDE statement for DFHPL1OI is included in the input to the linkage editor.

The major program management routines, those dealing with initialization/termination, error handling, and storage management are held together with a small control routine in the load module DFHSAP which is in the CICS nucleus. The routines differ from their equivalents in some respect and are differentiated by having a fourth letter of F. Thus within the load module DFHSAP there are three routines:

IBMFPIRA Initialization/termination under CICS

IBMFPGRA Storage management under CICS

IBMFERRA Error handling under CICS

Also included is IBMFPCCA, a control routine.

INITIALIZATION/TERMINATION

Initialization is caused by CICS passing control to DFHPL1OI, which, using the address in the CSA (a CICS control block), passes control to IBMFPCC, the control routine in DFHSAP. IBMFPCC then passes control to IBMFPIR which sets up the PL/I environment including the TCA, TIA, and CICS TCA appendage.

Under CICS, it is possible to specify a number of execution time options by using the character string PLIXOPT. If ISASIZE is specified in a PLIXOPT string, or through the default options IBMBXOPT, PL/I compares its minimum storage requirements to CICS's maximum. The minimum size required by PL/I is the sum of the Program Management Area and the main DSA. PL/I then issues a CICS GETMAIN command for the validated amount, which was rounded up to the next multiple of 8 bytes. If ISASIZE is not specified in anyway, then only the minimum request by PL/I is obtained.

The execution time options specified are examined by IBMFPIR and appropriate action is taken.

Options that can be specified are:

    COUNT
    FLOW
    HEAP
    ISAINC
    ISASIZE
    REPORT
    STAE

COUNT AND FLOW: Result in the setting up of space for COUNT and
FLOW output. They only take effect if the option was specified
during compilation. (COUNT can be specified at execution time
if FLOW was specified at compile time and vice-versa.)

HEAP: Is used to specify a separate storage area for CONTROLLED
and dynamically-allocated BASED variables. The maximum heap
size and increment, if specified, is either 65,496 bytes for
below 16 megabytes or 1,073,741,816 bytes above 16 megabytes.
The initial heap allocation and increments to heap are rounded
to the next higher multiple of 8 bytes.

ISAINC: Is used to obtain additional storage when a request for
LIFO storage cannot be satisfied from the ISA. PL/I uses the
larger of either the value specified in the ISAINC or the
amount of storage requested. The maximum ISAINC size is 65,496
bytes and is rounded to the next higher multiple of 8 bytes.

ISASIZE: Is used to specify the initial storage area that will
be acquired for PL/I storage.

REPORT: Is used to generate a report of storage usage. If it
is specified, a special storage handling module, IBMFPGD is
loaded.

STAE: Results in an EXEC CICS HANDLE ABEND command being
executed which leads to errors in PL/I being trapped by the PL/I
error handling.

When the environment is initialized, IBMFPIR passes control to
the PL/I program. After the PL/I program is completed, control
returns to IBMFPIR. A test is made to discover if any ON FINISH
statements were executed in the program and if they were, the
FINISH condition is raised. Finally, the PL/I acquired storage
is freed before control is returned to CICS.

CHAPTER 6. STORAGE MANAGEMENT

This chapter explains how the OS PL/I Optimizing Compiler allows you to regulate working storage. One of the principal methods of regulating working storage is through execution-time options.

The execution-time options, ISASIZE, ISAINC, HEAP, and TASKHEAP, all control working storage.

THE INITIAL STORAGE AREA

The initial storage area (ISA) is used for PL/I dynamic storage allocation. The start of the ISA contains a number of housekeeping fields known as the program management area. The program management area is described in Chapter 5, "Object Program Initialization" on page 74. For a diagram of the program management area, see Figure 33 on page 79. The remainder of the ISA is used for dynamic storage allocation which is described in this chapter. The ISA and its increments are always obtained below 16 megabytes.

By specifying ISASIZE, you set the amount of storage to be reserved for the ISA. The initialization routine, IBMBPII, issues a GETMAIN macro instruction to acquire the storage.

If you do not specify ISASIZE, the initialization routine takes the default action, which is to issue a variable GETMAIN instruction for the largest amount of contiguous storage possible. Half of this storage is allocated to the ISA, and the remainder is freed for possible future use by the program or by the operating system.

If you specify the ISASIZE to be greater than the region size, and you do not pass execution-time options as parameters at execution-time, your program will terminate with an 80A ABEND. If ISASIZE is greater than the region size, and execution-time options are passed as parameters, the initialization routine issues a variable GETMAIN for the largest amount of contiguous storage available, and uses the whole amount as the ISA.

The program initialization routine allocates the ISA. However, the allocation procedures for tasking and for CICS are slightly different. If you are using multitasking, see "Acquiring the ISA When Multitasking" on page 103; if you are using CICS, see "CICS Considerations" on page 103.

The value you specify in the ISAINC execution-time option is used to obtain additional storage when a request for LIFO storage cannot be satisfied from the ISA. For further details on how LIFO storage is allocated, see "Allocating and Freeing LIFO Storage" on page 89.

The values specified in the HEAP and TASKHEAP execution-time options are used to define separate storage areas for CONTROLLED and dynamically allocated BASED variables for the main (or only) task and for subtasks.

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 and for compiler-generated temporary values.

Dynamic storage can be conveniently divided into two classes:

1. That which is allocated and freed on a last-in/first-out (LIFO) basis.

   LIFO storage is also referred to as upper stack storage.

2. That which is not (non-LIFO storage).

   Non-LIFO storage, required by library modules for control blocks, is always allocated in the ISA. Non-LIFO storage for CONTROLLED or BASED variables can be allocated in the ISA or in a separate storage area called upper heap storage.

Contents of LIFO (Last-in/First-out) Storage

Two kinds of storage area are allocated in LIFO storage: dynamic storage areas (DSAs) and variable data areas (VDAs). A DSA is allocated for every procedure or block and contains:

- The system standard save area

- Certain standard housekeeping fields

- All automatic variables and compiler-generated temporaries whose length is known during compilation

A diagram of the standard section of a DSA is shown in Appendix A, "Control Blocks" on page 326.

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

- Storage for automatic variables and compiler-generated temporaries whose length is not known until execution. For example:

  DCL X CHAR (N);

- Workspace for certain library modules, such as storage for formatting PUT EDIT data.

- Allocations of library workspace (LWS) and on-control areas (ONCAs) after the occurrence of an interrupt.

- Storage for dynamic ON control blocks (ONCBs).

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 an automatic or static AREA).

- Workspace for certain library modules, such as control blocks for input/output and fetch.

DYNAMIC STORAGE ALLOCATION

When the HEAP execution-time option is not used, storage is allocated by the following principles:

- LIFO storage is allocated from the low-address end of ISA, starting at the first 8-byte boundary beyond the program management area.

- Non-LIFO storage is allocated from the high-address end of the ISA.

Between the areas of LIFO and non-LIFO storage is an unused
section known as the <u>major free area</u>. This area is shown in
Figure 34.

When heap storage is used, non-LIFO library workspace is still
allocated from the high-address end of the ISA. However,
program variables are allocated from the high-address end of a
separately acquired heap storage area. How storage is used in
the ISA when heap is specified is shown in Figure 35 on page 87.

```
                          Start of ISA
                      ┌─────────────────────────┐
                      │                         │
                      │   Program               │
                      │   Management            │
                      │   Area                  │
                 ┌─── ├─────────────────────────┤ ───             LIFO storage
                 │    │                         │    │
                 │    │   Main procedure DSA    │    │            Held in a contiguous
                 │    │                         │    │            stack, starting at the
                 │    ├─────────────────────────┤    ├─────       address following the
                 │    │                         │    │            program management
                 │    │   Subroutine DSA        │    │            area. Elements can be
 Area            │    │                         │    │            freed only from the
 Used for    ────┤    ├─────────────────────────┤ ───             high-address end of
 Dynamic         │    │                         │                 the stack.
 Storage         │    │   MAJOR FREE AREA       │
                 │    │                         │
                 │    ├─────────────────────────┤ ───             Non-LIFO storage
                 │    │                         │    │
                 │    │   3rd allocation for    │    │            Held in an upper stack
                 │    │   controlled variable C │    │            starting at the
 Head of ISA     │    ├─────────────────────────┤    │            high-address end of the
 free area───────┼──> │                         │    │            ISA. Any element in the
 chain           │    │                         │    ├─────       stack can be freed;
 (TLFE in TIA)   │    │   Space used by another │    │            consequently, all
                 │    │   controlled variable   │    │            elements are not
                 │    │   freed after 3rd       │    │            necessarily contiguous.
                 │    │   allocation            │    │            When elements are freed,
                 │    │   of C was made         │    │            the associated storage
                 │    │                         │    │            is placed on the
                 │    ├─────────────────────────┤    │            free-area chain and
                 │    │                         │    │            used for subsequent
                 │    │   2nd allocation for    │    │            allocations, when
                 │    │   controlled variable C │    │            possible.
                 │    ├─────────────────────────┤    │
                 │    │                         │    │
                 │    │   1st allocation for    │    │
                 │    │   controlled variable C │    │
                 └─── └─────────────────────────┘ ───
                          End of ISA
```

Figure 34. Use of Storage in the ISA if Heap Storage is Not Used

Start of ISA

```
┌─────────────────────────┐
│                         │
│      Program            │
│      Management         │
│      Area               │
│                         │
├─────────────────────────┤
│                         │      ┐
│   Main procedure DSA     │      │
│                         │      │
├─────────────────────────┤      ├──── LIFO storage
│                         │      │
│   Subroutine DSA         │      │
│                         │      ┘
├─────────────────────────┤
│                         │
│      MAJOR FREE AREA     │
├ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ┤
│                         │
│   Library Non-LIFO       │
│                         │
└─────────────────────────┘
```

Start of Heap Initial Storage

```
                                    ┌─────────────────────────┐
                                    │                         │  <┐
                                    ├─────────────────────────┤   │
                                    │   3rd allocation for     │   │
                                    │   controlled variable C  │   │
Head of Heap                        ├─────────────────────────┤  ─┘
free area
chain          ─────────>           │                         │
(TXLFE in                           │   Space used by another  │
TIA)                                │   controlled variable    │
                                    │   freed after 3rd allocation
                                    │   of C was made          │
                                    │                         │
                                    ├─────────────────────────┤
                                    │                         │
                                    │   2nd allocation for     │
                                    │   controlled variable C  │
                                    ├─────────────────────────┤
                                    │                         │
                                    │   1st allocation for     │
                                    │   controlled variable C  │
                                    └─────────────────────────┘
```

End of ISA

Non-LIFO Storage

Library non-LIFO storage is held in a stack that starts at the high-address end of the ISA. Programmer defined CONTROLLED and ALLOCATED BASED variables are held in a stack starting at the high-address end of the separate heap storage area. Any element in the stack can be freed; consequently, all elements are not necessarily contiguous. When elements are freed, the associated storage is placed on the free-area chain and used for subsequent allocations, when possible.

Figure 35. Use of Storage in the ISA if Heap Storage is Used

The last element in the LIFO stack is always freed first; consequently, it can always merge 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 appendage (TIA). Attempts are always made to use areas on this chain when further allocations of non-LIFO storage are made. When heap storage is used, PL/I maintains a separate free-area chain that is anchored in the TIA.

Before allocating LIFO storage, the major free area is tested for sufficient space. If there is not enough space, a new segment of the LIFO stack is obtained and the necessary

housekeeping fields are placed at its head. The largest area on the free-area chain is used as the new LIFO segment if it is large enough to satisfy the request.

If there is not enough space in the free area chain, the larger of either the value specified in the ISAINC or the amount of storage requested, is rounded to the next higher multiple of 4K bytes. A GETMAIN macro instruction is issued for this rounded amount.

Fields Used in Storage Handling

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

- The beginning-of-segment pointer (BOS)

- The end-of-segment pointer (EOS)

- The real end-of-segment pointer (TXRES)

- The next-available-byte pointer (NAB)

- The ISA non-LIFO free-area chain pointer (TLFE)

- A pointer to the byte beyond the end of the ISA (TISA)

- The heap storage initial address (TXHAD)

- The heap storage chain pointer (TXBOC)

- The heap free-area chain pointer (TXLFE)

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 X'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. This field is always zero if an additional segment of the LIFO stack is acquired, and the EOS value is held in TXRES in the TIA. When the REPORT option is in effect, this field is always zero.

The real end-of-segment pointer (TXRES) is used to store the pointer to the end of the major free area if an additional segment of LIFO storage was obtained or if the REPORT option is in effect. TXRES is held at offset X'60'(96) in the TCA appendage (TIA).

The next-available-byte pointer (NAB) is held in every DSA and points to the first 8-byte boundary contiguous with unused storage. This address is the start of the major free area. The current NAB is held in the most recent DSA. 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 first byte of BOS and NAB contains segment numbers ("00" for the ISA). The use of these numbers is explained under "Acquiring a New Segment of LIFO Storage" on page 93.

The ISA free area chain pointer (TLFE) The ISA non-LIFO free-area chain includes those elements of non-LIFO dynamic storage that were freed but that could not be merged with the major free area. The start of the chain is held at offset X'1C'(28) in the TCA appendage (TIA). TLFE points to the

element with the highest address. The ISA non-LIFO free-area chain is held in descending address order.

The pointer to the byte beyond the ISA (TISA) is used to keep track of the end of the ISA. TISA is held at offset X'0' in the TIA.

The heap storage initial address (TXHAD) is a pointer to the initial heap storage area. TXHAD is held at offset X'70'(112) in the TIA.

The heap storage chain pointer (TXBOC) includes all storage areas obtained by GETMAIN for use as heap storage. The start of the chain is held at offset X'74'(116) in the TIA. TXBOC points to the element with the highest storage address. The heap storage chain is held in descending address order.

The heap storage free-area chain pointer (TXLFE) includes those elements of heap dynamic storage that are not currently allocated. The start of the chain is held at offset X'78'(120) in the TIA. TXLFE points to the element with the highest storage address. The heap storage free-area chain is held in descending address order.

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. Allocation is handled the same way as the prolog code discussed in "Prolog" on page 32 under Chapter 2, "Compiler Output" on page 12. Freeing is done in the manner used by the epilog code, which is described in, "Epilog" on page 35 under Chapter 2, "Compiler Output" on page 12.

Before allocating LIFO storage, the major free area is tested for sufficient space for the new allocation. The test begins by adding the current value of NAB to the amount of storage that the DSA or VDA requires. The sum of those values is compared to the current value in EOS to determine whether there is enough space in the major free area.

If there is enough space, the sum becomes the new value of NAB, as shown in Figure 36 on page 90. This new value for NAB is the address for the byte beyond the end of the new allocation.

If there is not enough space in the major free area, PL/I calls the transient library storage management routine, IBMBPGR. The criteria for choosing the entry point depends upon whether a VDA or a DSA is being acquired. The process for obtaining LIFO storage is discussed under, "Acquiring a New Segment of LIFO Storage" on page 93. IBMBPGR sets BOS and TXRES to new values for the new segment. It also sets EOS to zero, so that all further LIFO requests go through IBMBPGR, and returns a new value of NAB to the compiled code or to the calling library routine.

When a DSA is being acquired, PL/I loads register 13 with the old NAB value (if space is available in the major free area) or with the address of the first byte past the storage management section of the new segment. The new NAB value is placed at offset X'4C' from register 13.

When a VDA is being acquired, the new NAB value replaces the old NAB value at offset X'4C' in the current DSA. A VDA is kept until it is explicitly freed by compiled code or until the block terminates. Freeing LIFO storage is done by restoring register 13 to its previous value when the block associated with the DSA terminates.

To Dummy DSA
```
A    R13──>┌─────────────────────────────────────────────┐  <──────┐
     │     ├─────────────────────────────────────────────┤         │
     └────>│  Back-chain (stored at fixed offset from R13)│         │
           ├─────────────────────────────────────────────┤         │
           │                                               │         │
           │              Main Procedure DSA               │         │
           │                                               │         │
           ├─────────────────────────────────────────────┤         │
           │  NAB (stored at fixed offset from R13)   ──────────┐    │
           ├─────────────────────────────────────────────┤    │    │
           │                                               │    │    │
    R13──> ├─────────────────────────────────────────────┤  <─┼────┘
  (=old NAB)│                                             │    │
           ├─────────────────────────────────────────────┤    │
           │                                               │    │
           │         Back-chain (=old R13)                 │────┘
           ├─────────────────────────────────────────────┤
           │                                               │
           │              Subroutine DSA                   │
           ├─────────────────────────────────────────────┤
           │              NAB                          ─────────┐
           ├─────────────────────────────────────────────┤     │
           │                                               │  <──┘
           ├─────────────────────────────────────────────┤
           │                                               │
           │              MAJOR FREE AREA                  │
           │                                               │
   EOS ──> └─────────────────────────────────────────────┘
```

┌──────────────────────────────────────┐   ┌──────────────────────────────────────┐
│ Allocating a new DSA   │   │ Freeing a DSA   │
├──────────────────────────────────────┤   ├──────────────────────────────────────┤
│ 1. Test if major free area large   │   │ 1. Load register 13 with current   │
│   enough for new DSA. If not,   │   │   back-chain address. Because   │
│   call IBMBPGRC.   │   │   the NAB and back-chain fields   │
│   │   │   are always addressed from   │
│ 2. Store R13 at fixed offset from   │   │   register 13, the previous   │
│   old NAB to act as back-chain.   │   │   values are automatically   │
│   │   │   restored.   │
│ 3. Load R13 with address of old   │   │   │
│   NAB. Within a segment of   │   │   │
│   LIFO storage, R13 for one DSA   │   │   │
│   is always equal to the NAB   │   │   │
│   of the previous DSA.   │   │   │
│   │   │   │
│ 4. Store new NAB at fixed offset   │   │   │
│   from register R13.   │   │   │
└──────────────────────────────────────┘   └──────────────────────────────────────┘

Figure 36.   Principles Involved in Allocating and Freeing LIFO Storage

ALLOCATING AND FREEING NON-LIFO STORAGE WHEN HEAP IS NOT USED

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 different method is therefore used. When storage that is contiguous with the major free area is freed, it is merged with the major free area by altering the end-of-segment (EOS) and TXRES pointers, which indicate 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 TIA. If the storage is contiguous to another block of storage already on the chain, the two are merged. 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 one of the four storage management library modules, whose address is held in the TCA. Figure 37 illustrates the principles involved. Whenever an allocation within the major free area is made, the end-of-segment (EOS) pointer in the TCA, or the real end-of-segment pointer (TXRES) in the TIA, is updated to point to the end of the major free area.

If there is not sufficient space in either the major free area or on the ISA free area chain, a GETMAIN macro instruction is issued for the required amount of storage. Non-LIFO storage acquired by a GETMAIN is freed by a FREEMAIN macro instruction.

NAB—>		NAB—>		NAB—>		NAB—>	
	Major Free Area		Major Free Area		Major Free Area		Major Free Area
		EOS—>		EOS—>			
			3rd Alloc.		3rd Alloc.		3rd Alloc.
EOS—>			2nd Alloc.		2nd Alloc.		2nd Alloc.
	2nd Alloc.			TLFE—>		TLFE—>	
	1st Alloc.		1st Alloc.				4th Alloc.

Initial situation	New allocation	Free 1st area	Further allocation
Two contiguous allocations of non-LIFO storage:	1. Free-area chain? No. (TLFE in TIA=0) 2. Allocate by altering EOS pointer.	1. Is area next to an area already on free-area chain? No. 2. Place area on free-area chain in descending address order.	1. Free-area chain? Yes. 2. Find first area that holds new allocation. Allocate at high-address end, leaving remaining area on the free-area chain. 3. Alter length at head of remaining area.

Figure 37. Principles Involved in Allocating and Freeing Non-LIFO Storage in the ISA

ALLOCATING AND FREEING HEAP NON-LIFO STORAGE

When the first ALLOCATE statement in the program is encountered, storage management issues a GETMAIN for the initial heap storage area. The larger of either the value specified on the HEAP execution-time option, or the amount of storage requested by the ALLOCATE, is rounded to the next higher multiple of 4K bytes. This rounded amount is used as the size in the GETMAIN request.

As in non-LIFO storage in the ISA, any increment of heap storage can be freed at any time. A separate heap free-area chain is maintained to keep track of available heap storage elements. When an allocation is made, the free-area chain is searched for an area large enough to hold the variable. If there is not sufficient space, a GETMAIN is issued to obtain an additional heap storage increment. The larger of the value specified in the HEAP execution-time option or the amount of storage requested by the ALLOCATE is rounded to the next higher multiple of 4K bytes. The rounded increment is used as the size on the GETMAIN request. Any unused space in the storage acquired is added to the heap free-area chain and is available for further allocations.

When heap storage is freed, it is placed on the free-area chain, which is anchored to the field TXLFE in the TIA. If the freed storage is contiguous with an element already on the free chain, the two are merged. If you have specified the FREE parameter on the HEAP option, the heap storage increment is freed by issuing a FREEMAIN macro instruction when the last variable in the increment is freed. The initial heap storage area is _not_ freed even if it becomes empty.

TXLFE>
```
┌─────────────┐
│             │
│   Free      │
│   Area      │
│             │
│             │
├─────────────┤
│             │
│ 2nd Alloc.  │
├─────────────┤
│ 1st Alloc.  │
└─────────────┘
```

TXLFE>
```
┌─────────────┐
│             │
│   Free      │
│   Area      │
│             │
├─────────────┤
│ 3rd Alloc.  │
├─────────────┤
│ 2nd Alloc.  │
├─────────────┤
│ 1st Alloc.  │
└─────────────┘
```

```
        ┌─────────────┐
        │             │
        │   Free      │
        │   Area      │
        │             │
        ├─────────────┤
        │ 3rd Alloc.  │
        ├─────────────┤
        │ 2nd Alloc.  │
TXLFE>  ├─────────────┤
        │   Free      │
        │   Area      │
        └─────────────┘
```

```
        ┌─────────────┐
        │             │
        │   Free      │
        │   Area      │
        │             │
        ├─────────────┤
        │ 3rd Alloc.  │
        ├─────────────┤
        │ 2nd Alloc.  │
TXLFE>  ├─────────────┤
        ├─────────────┤
        │ 4th Alloc.  │
        └─────────────┘
```

Initial situation	New allocation	Free 1st area	Further allocation
Two contiguous allocations of non-LIFO storage	1. Free-area Chain? Yes. 2. Allocate by altering length field at head of remaining area	1. Is area next to an area already on free-area chain? No. 2. Place area on free-area chain in descending order.	1. Free-area chain? Yes. 2. Find first area that holds new allocation. Allocate at high-address end, leaving remaining area on the free-area chain. 3. Alter length at head of remaining area.

Figure 38. Principles Involved in Allocating and Freeing Non-LIFO Storage in HEAP.

ACQUIRING A NEW SEGMENT OF LIFO STORAGE

PL/I internally tests the space between the NAB and the EOS to see if there is enough space for the DSA or VDA. The test is done every time a new procedure or block is entered, or when a VDA is required. If there is not enough space, an attempt is made to use the largest space on the ISA non-LIFO free-area chain as a new segment for the DSA or VDA.

If the largest space on the ISA non-LIFO free-area chain is not large enough for the DSA or VDA, a new area is obtained from the system by a GETMAIN macro instruction. The size of the area is the larger of either the value specified in the ISAINC execution-time option, or the space required for the DSA or VDA. The size is rounded up to the next multiple of 4K bytes.

The former values of BOS and TXRES are stored at the start of the new segment. Pointers are set as follows:

• BOS in the TCA, is set to point to the beginning of the new segment.

• EOS in the TCA, is set to 0.

• TXRES in the TIA, is set to point to the end of the new segment.

The DSA or VDA is allocated storage 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.

When a new DSA or VDA is required after a second LIFO segment
was required, EOS is less than the sum of NAB and the required
length, because EOS is now zero. Consequently, it appears that
there is insufficient space for the DSA or VDA in the segment,
regardless of whether or not this is the case. The library
module, IBMBPGR, is called to allocate the new DSA or VDA.
IBMBPGR also checks for empty segments and restores BOS and
TXRES (and EOS if the first segment is now the current segment.)
If there is space for the DSA or VDA in the current segment,
IBMBPGR adds the empty segment to the free-area chain when the
segment is in the ISA. Otherwise, it frees it with a FREEMAIN
instruction if it is an ISA increment. The process is
illustrated in Figure 39.

Each segment receives a segment number, starting at hexadecimal
"00" and decreasing by 1 for each new segment. The number for
the first segment is "00", the second segment "FF", and so on.
This number is held as the first byte of the NAB and BOS
pointers.
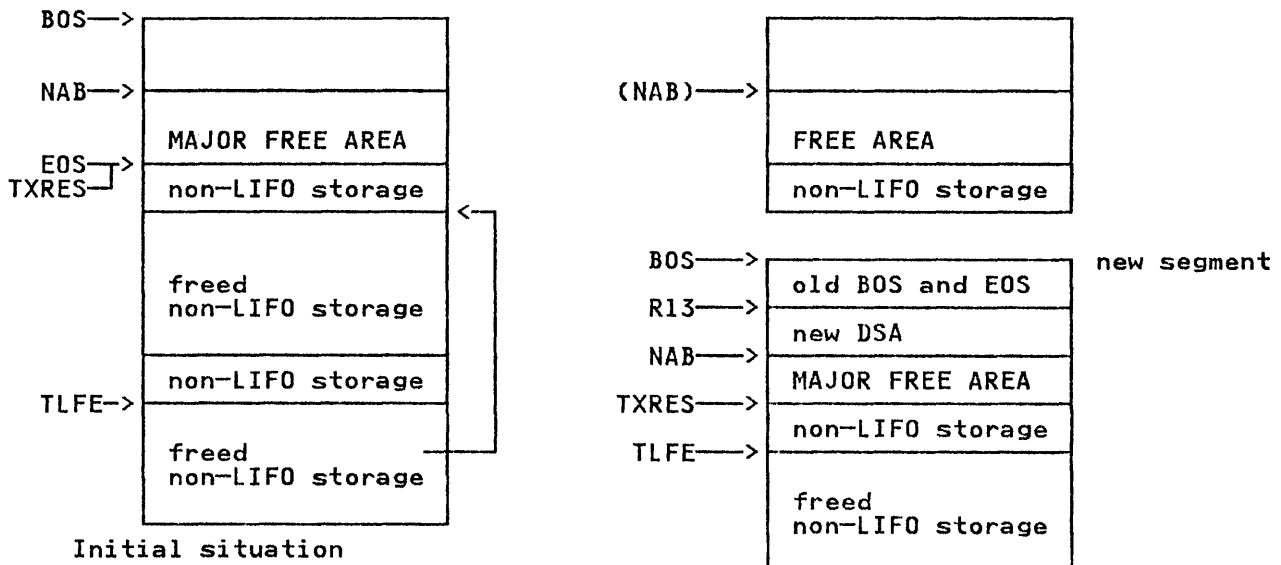
Initial situation

1. Free area chain exists.
   BOS, NAB, and EOS have
   X'00' in the
   first byte, i.e.,
   segment number 1.

Acquiring new segment

1. Compiled code or library routine
   finds the major free area too small.
   Calls IBMBPGR.

2. IBMBPGR finds an area on the free
   area chain large enough for
   allocation.

3. Stores old BOS and EOS. Sets new
   BOS and TXRES, zeros EOS and
   returns to caller.

4. Caller gets new DSA. BOS and
   NAB have X'FF' in the first
   byte, i.e., segment number 2.

Figure 39 (Part 1 of 2).   Principles Involved in Allocating and Freeing Segments of
PL/I Dynamic Storage

```
                                      BOS──>┌──────────────────┐
NAB──>┌──────────────────┐                  │                  │
      │                  │           R13──>├──────────────────┤
      │   FREE AREA      │                  │   New DSA        │
      ├──────────────────┤           NAB──>├──────────────────┤
      │  non-LIFO storage│                  │  MAJOR FREE AREA │
BOS──>├──────────────────┤           EOS─┐ ├──────────────────┤
      │  old BOS and EOS │           TXRES┘ │  non-LIFO storage│
      ├──────────────────┤                  ├──────────────────┤        <┐
      │                  │                  │                  │         │
      │  empty segment   │                  │   free           │         │
      │                  │                  │   non-LIFO storage│        │
TXRES─>├──────────────────┤                  ├──────────────────┤        │
      │  non-LIFO storage│                  │  non-LIFO storage│        │
TLFE──>├──────────────────┤           TLFE──>├──────────────────┤        │
      │                  │                  │                  │         │
      │  free            │                  │   free           │         │
      │  non-LIFO storage│                  │   non-LIFO storage│<───────┘
      └──────────────────┘                  └──────────────────┘
```
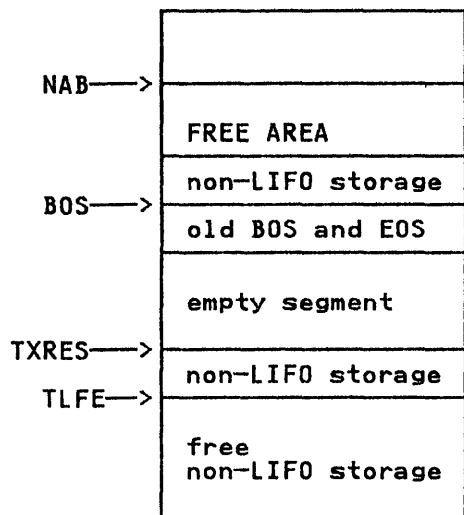
Freeing DSA in segment	Freeing segment

1. Register 13 is restored
   in the normal way. BOS, EOS
   and TXRES are not restored.
   The segment is not freed
   until there is a further
   demand for storage.

2. NAB now has X'00' in
   the first byte, BOS and
   TXRES still have X'FF.',
   and EOS is still 0.
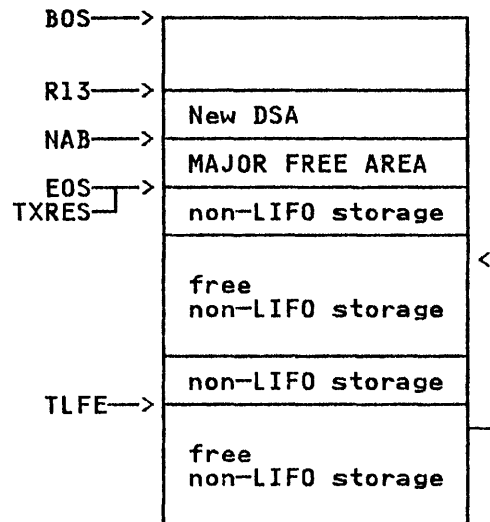
1. When storage is again required,
   NAB + storage required is compared
   with EOS.

2. NAB + storage is found to be
   greater (because EOS=0),
   so IBMBPGR is called.

3. IBMBPGR finds the segment numbers
   are different. It tests to see if
   new storage fits in the old segment
   If not, it allocates it in the
   current segment.

4. Storage fits, so restore the old
   BOS, EOS, and TXRES, place the
   segment on the free-area chain,
   and return to the caller.

5. Caller allocates storage starting
   at the current NAB.

Figure 39 (Part 2 of 2). Principles Involved in Allocating and Freeing Segments of
PL/I Dynamic Storage

STORAGE MANAGEMENT ROUTINES

Depending upon the characteristics of your programs, one of the
following six storage management library modules is selected:

* IBMBPGR, for non-tasking, NOREPORT

* IBMBPGD, for non-tasking, REPORT

* IBMFPGR, for CICS, NOREPORT

* IBMFPGD, for CICS, REPORT

* IBMTPGR, for multitasking, NOREPORT

* IBMTPGD, for multitasking, REPORT

Each of these modules performs similar functions. The following
description of IBMBPGR applies to all six modules, with the
exceptions noted under "Storage Reports" on page 98,

"Multitasking Considerations" on page 103 and "CICS Considerations" on page 103.

The allocation and freeing of LIFO storage within the first segment are handled by compiled code or by the library module requiring the storage. All other dynamic storage allocation is carried out by the transient library routine, IBMBPGR; this module has four entry points:

IBMBPGRA   Allocate non-LIFO storage.

IBMBPGRB   Free non-LIFO storage.

IBMBPGRC   Obtain and free additional storage segments (for DSAs).

IBMBPGRD   Obtain and free additional storage segments (for VDAs).

These entry points are described below. In all cases, storage is allocated in multiples of 8 bytes.

Allocating Non-LIFO Storage (Entry A)

When entered from entry point IBMBPGRA, the module first searches the ISA non-LIFO free-area chain, or the heap free-area chain (if one exists), and allocates storage in the first area large enough to hold the request. If heap storage is not used and there is no chain, or if no area on the chain is large enough, IBMBPGR 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, a GETMAIN macro is issued for the required storage.

If heap storage is used and there is no chain, IBMBPGR checks for an initial heap allocation. If none exists, it issues a GETMAIN for the initial heap storage area. When heap initial storage exists and there is no chain, or when no area on the chain is large enough, IBMBPGR issues a GETMAIN for a heap storage increment. For an overview of how PL/I allocates heap storage, see "Allocating and Freeing Heap Non-LIFO Storage" on page 92

If the GETMAIN cannot be satisfied, the system ends the job with either an ABEND-code 80A or 878. This ABEND is intercepted by the ABEND analyzer, IBMBPES. IBMBPES issues a message indicating which statement was being executed and when the demand for storage was made. It then returns to the system to complete the ABEND.

Provided that storage can be allocated, control returns to the caller, with register 1 pointing to the address of the storage allocated.

Freeing Non-LIFO Storage (Entry B)

When freeing non-LIFO storage, or segments of LIFO storage IBMBPGR first tests to discover whether the element being freed is within the ISA. This test is done by seeing if the address is between the value held in register 12, the address of the TCA, and the value held in the TISA field of the TIA, which points to the end of the ISA. If the element is outside the ISA and heap storage is not used, it was acquired. It is therefore freed with a FREEMAIN macro instruction.

When the element to be freed is within the ISA, the module scans the ISA non-LIFO free-area chain (if one exists) to see whether the storage being freed can be merged with areas already on the chain. This is done if possible. The module then determines whether the storage being freed is adjacent to the major free area. If so, EOS is changed to point to the end of the area being freed, or to the end of the merged area, if this adjoins

the major free area. If the element cannot be merged with any
other, the area is added to the ISA non-LIFO free-area chain,
which is arranged in descending order of addresses.

If heap storage is used, the heap free-area chain (if one
exists) is scanned to determine whether the storage being freed
can be merged with areas already on the chain. This is done if
possible. If the element cannot be merged with any other, the
area is added to the heap free-area chain, which is arranged in
descending order of addresses. The format of a free area chain
element is shown in Figure 40. If the HEAP FREE option is
specified, IBMBPGR tests whether the heap storage increment is
empty. If so, the increment is freed with a FREEMAIN macro
instruction. The heap initial storage area is not freed even if
it becomes empty.

Segment Handling (Entry C and Entry D)

IBMBPGR is called when compiled code finds that the address in
the pointer NAB, plus the length of the new DSA or VDA to be
allocated, is greater than the value of the pointer EOS.
IBMBPGR is called either at entry point C or entry point D,
depending on whether the storage is required for a DSA or for a
VDA. Entry point C is used if a DSA is required; entry point D,
if a VDA is required. The difference is the method used to
store the caller's registers. IBMBPGRC stores the caller's
registers in a special save area in the TCA and TIA,
respectively, because no DSA has yet been acquired. IBMBPGRD
stores the registers in the caller's DSA, in the usual manner.

IBMBPGRC and IBMPGRD check to see if the number in the first
byte of NAB is greater than the number in the first byte of
TXRES. If the difference is greater than one, more than one
extra segment was allocated for DSAs or VDAs that are no longer
current. In this case, PL/I frees segments until only one empty
segment remains. It does this by setting BOS and TXRES 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, PL/I tests whether the new
DSA fits into the segment that contains the present NAB pointer
(the segment before the empty segment). This test compares the
current NAB pointer with the old TXRES 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. Control returns to the caller, with a new value for
TXRES and BOS. The DSA is allocated immediately after the old
NAB.

0	Length of element in bytes
4	Pointer to area with lower address, zero if last element
8	Unused storage
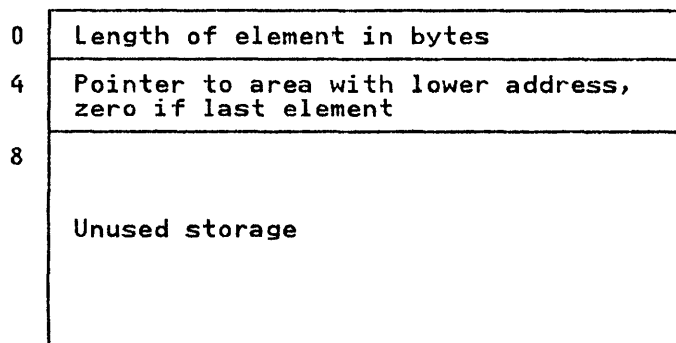
Figure 40. Format of Element on Free-Area Chain

If there is not enough room in the segment containing NAB, PL/I
determines whether the empty segment is large enough to hold the
new DSA. This check is done by comparing the difference between
the current BOS and TXRES 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 storage is passed to compiled code in general register 1, and the address of the new NAB is passed in general register 0.

If there is not enough room in the empty segment, the segment is freed. There are now no empty segments, and the situation is treated as if there were never any empty segments.

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

If the segment numbers are the same, a check is made to see whether the new DSA or VDA fits in the current segment. (Apparent overflow was caused by EOS=0.) This check is done by comparing the sum of the value in the NAB and the length of the DSA or VDA with the value in TXRES. If there is space in the current segment, control is returned to the caller.

If there is not enough space in the current segment (true overflow), a new segment must be allocated. A new segment is allocated by searching the ISA non-LIFO 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, the larger of either the value specified on the ISAINC execution-time option, or the requested amount, is rounded to the next higher multiple of 4K bytes. That value is then used in a GETMAIN macro instruction. The new segment is set up in the area acquired.

When a new segment is allocated, the old values of BOS and TXRES are placed in control words at the head of the new segment. New values for BOS and TXRES, which point to the beginning and end of the new segment with first byte numbers 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. The format of a secondary segment is shown in Figure 41.

```
                                      <--current
  0  | Previous BOS value    |          BOS
     |-----------------------|
  4  | Previous TXRES value  |
     |-----------------------|
  8  |                       |
     |                       |
     | DSA or VDA            |
     |                       |
     |                       |
     |-----------------------|
     | Major Free Area       |
     -------------------------  <--current
                                      TXRES
```
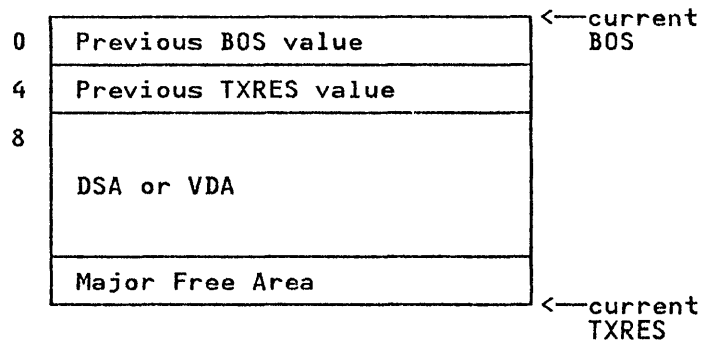
Figure 41.   Format of Second and Subsequent Segments of the LIFO Stack

STORAGE REPORTS

When you request a storage report, you are given a report after the program completes that shows:

• The ISA size specified (if a size was specified).

• The ISA increment size specified (if ISAINC was specified).

• The ISA size used.

• The amount of PL/I storage required by the program. (This is a suggested optimum ISA size.)

- The maximum amount of storage obtained outside the ISA at any one time.

- The number of stack GETMAIN macro instructions issued.

- The number of stack FREEMAIN macro instructions issued.

- The number of requests to acquire ISA non-LIFO storage.

- The number of requests to free ISA non-LIFO storage.

- The heap size specified (0 if HEAP is not specified).

- The heap increment specified. 4096 bytes appears if a heap increment is not specified.

- The amount of heap storage required. This is the maximum amount obtained at any one time.

- The number of heap GETMAIN macro instructions issued.

- The number of heap FREEMAIN macro instructions issued.

- The number of requests to obtain heap non-LIFO storage.

- The number of requests to free heap non-LIFO storage.

The report is generated by the storage report routine, IBMBPGD. This module is loaded during program initialization, instead of the normal storage management module, IBMBPGR. IBMBPGD has the same entry points and carries out the same functions as IBMBPGR. (For further details on IBMBPGR, see "Storage Management Routines" on page 95.) However, it also maintains a record of certain storage statistics. To ensure that IBMBPGD handles all storage allocation both inside and outside the ISA, the EOS field in the TCA is set with a dummy value of zero. The dummy value is set so that the storage routine will be called whenever LIFO storage is required, as well as for non-LIFO storage and stack overflow requests.

The storage report is issued during program termination. The termination routine, IBMBPIT, calls the report writing module, IBMBPMR. The report is transmitted to the dump file.

Action during Initialization

During program initialization, if REPORT was included in the parameters passed to IBMBPIR, the report storage management routine, IBMBPGD, is loaded, and its entry point addresses are placed in the TCA. The value in the end-of-segment pointer, EOS, is set to zero. Space for a report table is acquired, and the true value of the end of segment is placed in TXRES in the TIA.

Action during Execution

During execution, IBMBPGD is called each time there is a request for PL/I dynamic storage. It is called for non-LIFO storage in the normal way, and, when LIFO storage is required, it is called because the zero value in EOS results in the value of NAB+DSA or VDA being greater than EOS. Consequently, the stack overflow routine (IBMBPGD, entry point C or D) is called. When a call is made to entry point C or D, IBMBPGD makes a test against the true value of the end of segment held in TXRES, and, if there is sufficient room, the storage is acquired in the current segment of the LIFO stack. If there is not sufficient room, IBMBPGD takes the same action as IBMBPGRA, which is described in "Allocating Non-LIFO Storage (Entry A)" on page 96.

All other storage acquisition by IBMBPGD is handled in exactly the same way as for the corresponding entry point of IBMBPGR.

However, IBMBPGD keeps a running total of the following in the
storage report table.

1. The highest value obtained by subtracting the current length
   of the major free area from the current amount of PL/I
   storage acquired outside the ISA.

2. The largest amount of PL/I storage obtained outside the ISA
   at any one time.

3. The number of stack GETMAIN macro instructions issued.

4. The number of stack FREEMAIN macro instructions issued.

5. The number of requests to acquire ISA non-LIFO storage.

6. The number of requests to free ISA non-LIFO storage.

7. The highest value obtained by adding the lengths of all heap
   storage segments allocated at one time.

8. The number of heap GETMAIN macro instructions issued.

9. The number of heap FREEMAIN macro instructions issued.

10. The number of requests to obtain heap non-LIFO storage.

11. The number of requests to free heap non-LIFO storage.

The values are altered if necessary every time IBMBPGD is
entered. The value of 1 and 2 above is calculated on every
call, and the highest number retained in the report table. The
format of the storage report table is given in
Appendix A, "Control Blocks."

Action on Termination

On termination, the termination routine, IBMBPIT, calls the
storage report writing module, IBMBPMR, which transmits the
storage report onto the dump file.

The amount of PL/I storage required is calculated by adding the
figure described in 1 to the ISA size used. The figure will be
positive if any storage outside the ISA was acquired; it will be
negative or zero if no storage was acquired outside the ISA.

Two items should be noted about the results produced by a
storage report.

1. If storage was acquired outside the ISA, the figure given
   for storage used cannot be taken as final. A further
   request for a report when the program is run in the ISA size
   suggested may result in a smaller figure being generated.
   This smaller size should be used. This discrepancy is
   caused by the differences in acquiring storage inside and
   outside the ISA. To obtain a correct figure using only one
   run, the program should be run in a large ISA that can be
   expected to hold all PL/I storage.

2. The report can only refer to the particular run of the
   program on which the report was given. Runs with different
   data or parameters may have different storage requirements.

Storage Reports for Multitasking Programs

Storage reports for multitasking programs are generated in the same way as those for non-multitasking programs. A special storage management module (IBMTPGD), is loaded at execution time, and retains statistics on the amount of storage used. To ensure this module handles all requests for storage, the value in EOS is set to zero, and the true EOS value is retained in TXRES in the TIA. The report is issued during program termination by module IBMBPMR.

For a multitasking storage report the following information applies.

FOR THE MAJOR TASK: The same as for a non-multitasking program (see above).

FOR SUBTASKS, A COMBINED REPORT FOR ALL SUBTASKS SHOWING

The subtask ISA size specified (if a size was specified)

The subtask ISA increment size specified (if ISAINC was specified)

The maximum ISA size used by any subtask

The minimum ISA size used by any subtask

The maximum PL/I storage required by any subtask

The minimum PL/I storage required by any subtask

The maximum amount of storage acquired outside the ISA by any subtask

The minimum amount of storage acquired outside the ISA by any subtask

The maximum amount of heap storage acquired by any subtask

The minimum amount of heap storage acquired by any subtask

The total number of stack GETMAIN and FREEMAIN macro instructions issued by all subtasks

The total number of requests to acquire and free ISA non-LIFO storage issued by all subtasks

The subtask heap size specified (0 if TASKHEAP was not specified)

The subtask heap increment size specified (4096 bytes if TASKHEAP increment was not specified)

The total number of heap GETMAIN and FREEMAIN macro instructions issued for all subtasks

The total number of requests to acquire and free heap non-LIFO storage issued by all subtasks

To enable these figures to be produced, a multitasking version of the storage report module is used. This module, IBMTPGD, has two more entry points than its non-multitasking counterpart. These are:

IBMTPGDE   Called when a task is initialized.

IBMTPGDF   Called when a task is terminated.

IBMTPGDE is called when a task is initialized. It acquires storage for the report table for the task, and retains a record of the number of active PL/I tasks, increasing the maximum number if necessary.

IBMBPGDF is called when a task is terminated. If the terminated task is a subtask, IBMBPGDF completes the relevant fields in the subtask storage report table, from information in the report table of the terminating task.

During initialization, space is required by the control task for a combined subtask report table. The information in this report table is used to generate the merged subtask report. During the initialization of each task, space for a report table for that task is obtained. The report table for the major task is flagged.

Throughout the execution of each task, a separate report table is maintained. At the end of each subtask, the information in the terminating task is merged into the combined subtask table, held in the storage associated with the control task.

When the jobstep is terminated, IBMBPMR produces the information from the merged subtask report table and the report table of the major task. (IBMBPMR is used to output the report for both tasking and non-multitasking programs.)

STORAGE MANAGEMENT IN PROGRAMMER-ALLOCATED AREAS

By using area variables, you can obtain 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 are handled by the library module, IBMBPAM.

IBMBPAM keeps a check on the amount of storage allocated. If there is not enough space for an allocation, or if the target area is too small to hold the source area 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 in the 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.

Before a space is freed, a check is made to see whether it is contiguous with a space or spaces that are already on the free storage chain. If it is, the contiguous spaces are merged. A check is then made to see whether the amalgamated space is contiguous with the OEE. If the space is contiguous with the OEE, the OEE is pointed to the start of the space, and the space removed from the free storage chain. If the merged space is not contiguous with the OEE, the free area chain is rearranged so that it is in the correct order.

If the space to be freed is not contiguous with another space on the free storage chain, a check is made to see if it is contiguous with the OEE. If it is, the OEE is updated.

If the space to be freed is contiguous neither with the OEE nor with another space on the free storage chain, the space is placed in its correct position in the storage chain.

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.

MULTITASKING CONSIDERATIONS

Storage handling within each task follows the pattern described above, except that certain storage requests are made for storage that will be available to all tasks. This storage has to be obtained in subpool 0. To indicate such a requirement, IBMTPGR is called with a negative value. A GETMAIN for the specified amount is then issued to subpool 0, a negative value indicating that the storage must be in subpool 0.

The method used to acquire the ISA is slightly different for tasking. This is described below.

Acquiring the ISA When Multitasking

The size of the ISA required for the major task and every minor task can be requested in the ISASIZE execution-time option. If the size specified in the parameter is smaller than that needed for the program management area, only the exact size required for the program management area is acquired and all further allocations of dynamic storage are made by issuing GETMAIN macro instructions. These allocations are made in exactly the same way as they are when non-multitasking programs cannot acquire space within the ISA. (See the discussion under "Storage Management Routines" on page 95.)

The default action, taken if no ISA size is specified, is to use the larger of:

• The installation default for ISASIZE

• The storage for the program management area and the DSA for the main procedure.

The storage requirement is rounded up to the next 4K-byte increment before the GETMAIN is issued.

The IBM-supplied installation default for ISASIZE in a tasking environment is 8K bytes. This size is usually enough to hold the program management area and the DSA for the main procedure. An exceptionally large PRV or large automatic storage requirements might cause the ISA to be larger than 8K.

CICS CONSIDERATIONS

Storage management under CICS is handled by a module called IBMFPGR. It is based on the OS storage management module IBMBPGRA. As with the system described in "Allocating and Freeing LIFO Storage" on page 89, an area of storage called the ISA is acquired and program management blocks are placed at the low-address end. This occurs during program initialization. During actual execution, block dependent storage (mostly automatic variables and housekeeping fields) is placed in a LIFO (last-in/first-out) stack immediately following the program management blocks. Any storage that will not be freed on a last-in/first-out basis is put at the other end of the ISA, or in heap storage in a separate non-LIFO stack. (Such storage is nonblock dependent such as BASED variables.)

Thus the majority of storage requests can be met without requests to the CICS system. There are, then, two stacks within the ISA encroaching on a free area known as the major free area (see Figure 42 on page 104). There is never any problem of unused space in the LIFO stack, however, there may be in the non-LIFO stack and accordingly a free area chain is kept of any free areas and when non-LIFO storage is being allocated, an attempt is made to use these spaces.

STORAGE AVAILABLE TO CICS

```
┌──────────────────────────────────────────────────────────────┐
│                                                                │
│  ISA                       ISAINC                              │
│  Acquired from CICS        Additional Segment                  │
│                            acquired from CICS                  │
│  ┌─────────────────────┐   ┌ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ┐          │
│  │                     │                                       │
│  │  Program            │   │ Acquired when there is │          │
│  │  Management         │     not enough room for a             │
│  │  Blocks             │   │ section of storage in  │          │
│  │                     │     the major free area or            │
│  ├─────────────────────┤   │ in an empty space in   │          │
│  │                     │     the non-LIFO stack.               │
│  │  LIFO stack         │   │                        │          │
│  │                     │                                       │
│  │                     │   └ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ┘          │
│  ├─────────────────────┤                                       │
│  │                     │   HEAP                                 │
│  │                     │                                       │
│  │  Major Free Area    │   ┌─────────────────────────┐         │
│  │                     │   │                         │         │
│  ├─────────────────────┤   │ Acquired when the HEAP  │         │
│  │                     │   │ execution-time option   │         │
│  │  Non-LIFO stack     │   │ is specified for BASED  │         │
│  │  (may contain       │   │ or CONTROLLED           │         │
│  │  empty spaces)      │   │ variables.              │         │
│  │                     │   │                         │         │
│  └─────────────────────┘   └─────────────────────────┘         │
│                                                                │
└──────────────────────────────────────────────────────────────┘
```

Figure 42. Storage Management under CICS

When there is no room in the ISA for LIFO requests, additional
storage is acquired from CICS. PL/I determines the amount
needed by taking the larger amount of either, the amount
currently requested or, the amount specified on ISAINC This is
called a new segment which is freed as soon as its contents are
discarded.

Likewise, when there is no room in the ISA or ISAINC for
non-LIFO requests, additional storage is acquired for the amount
requested from CICS, unless the HEAP option is specified. If
the HEAP option is specified, the acquired storage for non-LIFO
request becomes the larger of either the amount requested or the
amount of heap storage specified.

When the REPORT execution time option is used, another storage
management routine IBMFPGD is called. This keeps track of the
high and low water marks of the stacks and the amount of storage
acquired outside the ISA and allows the user to determine their
storage needs and specify them in the execution-time ISASIZE
option. Execution-time options are described in the OS PL/I
Optimizing Compiler: Programmer's Guide.

This chapter deals with the method used to implement <u>execution time</u> error handling. All errors detected at execution time are associated with PL/I conditions and can be handled either by ON-units written by the programmer or by standard system action, as defined by the PL/I language.

The chapter starts with a brief discussion of the terms and concepts used in error handling. A discussion of the error handling facilities offered by the operating system and those specified in the PL/I language follows. The implementation problems these facilities raise and the method used to solve them are then described. A separate section is devoted to the CHECK condition because this raises special problems. The chapter is completed by a brief discussion of the error message modules, the modules used to implement the PLIDUMP facility, and the handling of the compiler FLOW option.

Error detection during compilation is not covered in this chapter. Nor is any advice given on how to use PL/I error handling facilities. Advice on debugging with dumps is given in Chapter 12.

Note: If the NOSPIE or NOSTAE options are specified in the parameters for the procedure, much of what is said in this chapter does not apply. The PL/I SPIE/ESPIE or STAE/ESTAE macros are not issued and system-detected interrupts and ABENDs are not handled in the PL/I defined manner.

Terminology

Throughout this chapter a number of special terms are used. Some of them are terms used in the PL/I language, others are terms that are used to describe certain implementation features and concepts. The terms are listed below.

ESTABLISHED: This term is used to describe ON-units and, sometimes, ON statements. The ON-unit or statement is said to be <u>established</u>, if the action specified in the ON-unit or ON statement will be taken should the specified condition arise. Thus an ON-unit becomes established when the ON statement is executed and ceases to be established when the ON or REVERT statement referring to the same condition is executed, or when the associated block is terminated.

ENABLED: This term is used to describe certain PL/I conditions (SIZE, CONVERSION, etc.). A condition is enabled when the occurrence of the condition will result in the execution of an ON-unit or standard action. A condition is disabled when the occurrence of the condition will, apparently, be ignored.

QUALIFIED AND UNQUALIFIED CONDITIONS: Qualified conditions are those conditions, such as ENDPAGE, that need to be qualified by a file or other name. Unqualified conditions are those that do not need qualification. Figure 45 on page 108 shows which conditions are qualified and which are unqualified.

PROGRAM CHECK AND SOFTWARE INTERRUPTS: Certain PL/I conditions are detected automatically by the computing system. Others have to be detected by special checking code either in the library modules or in the compiled program. Interrupts detected by the system are referred to as <u>program check</u>. Interrupts detected by special checking code are referred to as <u>software detected</u> or <u>software interrupts</u>. A list of program check interrupts and their associated PL/I conditions are given in Figure 43.

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

Figure 43. Machine Interrupts Associated with PL/I Conditions

These terms program check and software interrupts are used for convenience in this publication and are not accepted terms in the PL/I language. Figure 45 on page 108 shows which interrupts are system detected and which are software detected.

STATIC AND DYNAMIC DESCENDENCY: Static and dynamic descendency are terms used to define the scope of the PL/I features. ON-units are dynamically descendent. That is, they are inherited from the calling procedure in all circumstances. Condition enablement is statically descendent. That is, it is inherited from the containing block in the source program. Static descendency can be determined during compilation. Dynamic descendency cannot be know until execution. See Figure 44 on page 107.

NORMAL RETURN: Normal return is return from a called block by means of reaching the END or RETURN statement rather than because of a GOTO out of the block. In an error-handling context, normal return is taken to mean normal return from the ON-unit. The action taken after normal return from an ON-unit is specified in the PL/I language. For most conditions, it is to return to the point of interrupt.

STANDARD SYSTEM ACTION: Standard system action is the name given to the default PL/I-defined action taken when a condition occurs and there is no established ON-unit for that condition.

BACKGROUND TO ERROR HANDLING

System Facilities

The operating system offers certain error-handling facilities. These can be summarized as follows:

Various situations can cause a machine interrupt which results in entry to the supervisor. It is possible for the programmer to define the action that will be taken after any of these interrupts by means of a routine specified in a SPIE/ESPIE macro instruction. Alternatively, the programmer can accept the default action of the system. It is also possible for the programmer to prevent the occurrence of certain interrupts by masking out fields in the PSW.

```
(SIZE):B:PROC;

ON ERROR SNAP;
•
•
•
CALL C;


        C:PROC;



CALL D;
```

```
    D:PROC;             E:PROC;
    •
    •
    •
    CALL E;
```
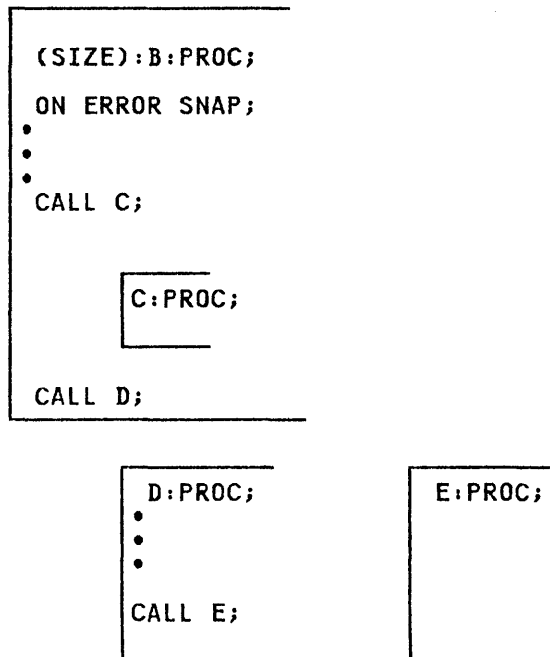
Static descendency: the enablement prefix (SIZE): in procedure B
is inherited only by the contained procedure C, regardless of
which procedure calls which.

Dynamic descendency: the ON-unit ON ERROR SNAP; is inherited by
any procedure called by B and any subsequently called
procedures. Thus, if B calls D, which calls E, the ON-unit is
established in procedure E.

Figure 44. Static and Dynamic Descendency

PL/I FACILITIES

The PL/I language offers similar but greatly extended
facilities. The number of situations causing interrupts is
considerably larger and some, such as ENDFILE, can be used to
control normal program flow rather than to handle errors. The
use of ON-units allows the programmer to obtain control after
most interrupts.

Alternatively the programmer can accept standard system action.
The programmer also has the choice of whether certain conditions
will cause interrupts. This is done by enabling or disabling
the conditions. If the condition is disabled, neither ON-unit
nor standard system action will be taken if the condition
occurs.

A number of PL/I conditions correspond directly to the
interrupts that are detected by the operating system (see
Figure 43 on page 105). Other conditions however belong only to
PL/I.

The majority of PL/I conditions are caused by errors in program
logic or the data supplied. Some, however, are not connected
with errors. These are conditions such as ENDFILE, which occur
at unpredictable times and consequently cannot be easily
anticipated by code in the source program.

Conditions that are most probably caused by programming errors
are know as error conditions. Figure 45 on page 108 shows which
conditions are error conditions. The standard system action for

these conditions is to put out a message and raise the ERROR
condition.

The ERROR condition is also raised by any programming error that
is not directly covered by a PL/I condition. A data interrupt,
for example, raises the ERROR condition, and certain software
detected conditions, such as taking the square root of a real
negative number, also raises the ERROR condition.

The ERROR condition consequently gives the programmer blanket
coverage of all program errors. The ERROR condition differs
from all other conditions in that a diagnostic message is always
generated regardless of whether an ERROR ON-unit exists. If an
ERROR ON-unit exists, the message is generated before ON-unit
action is taken.

A further facility offered by PL/I is the availability of
condition built-in functions and pseudo-variables. These allow
the programmer to inspect various fields associated with the
interrupt and, in certain cases, to alter the contents of these
fields.

The situation in PL/I is complicated by the question of the
scope of ON-units and condition enablement. Condition
enablement is statically descendent and can be decided during
compilation. ON-units, however, are dynamically descendent and
the establishment or otherwise of ON-units can only be decided
during execution. (See "Terminology" on page 105).

Name of Condition	Quali- fied	Description	Recognized By	Default	Pro- grammer Control	ERROR[2] Condi- tion
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 hardware	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

Figure 45 (Part 1 of 3). PL/I Conditions

Name of Condition	Quali-fied	Description	Recognized By	Default	Pro-grammer Control	ERROR[2] Condi-tion
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
Program Checkout						
SUBSCRIPTRANGE	no	Array subscript outside declared bounds	Compiler-generated checking code	disabled	yes	yes
STRINGSIZE	no	Attempt to assign a string of too great length	Code in relevant library modules	disabled	yes	no
STRINGRANGE	no	Attempt to access beyond limits of string	Code in relevant[1] library modules	disabled	yes	no
CHECK (variable or label)	yes/no	Value assigned to identifier or control passed through label	Compiler-generated checking code, or library module	disabled	yes	no

Figure 45 (Part 2 of 3). PL/I Conditions

Name of Condition	Quali-fied	Description	Recognized By	Default	Pro-grammer Control	ERROR[2] Condi-tion
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[2]	Relevant library modules, compiled code, or system	enabled	no	-
FINISH	no	Program ending	Compiled code	enabled	no	-
Programmer Named						
CONDITION (name)	no	Programmer defined condition	Signal statement	enabled (when coded)	no	-
Conversational						
ATTENTION	no	Attention interrupt occurs	System + compiled code	disabled	yes	no

Figure 45 (Part 3 of 3). PL/I Conditions

Notes to Figure 45:

[1]   When STRINGRANGE is enabled, appropriate library modules are always called.

[2]   The ERROR condition is raised when an error occurs that is not covered by PL/I exceptional conditions. 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.

IMPLEMENTATION OF ERROR HANDLING

To implement the PL/I error handling scheme it is necessary to be able to detect all the PL/I conditions, to acquire various information about how the conditions occurred for condition built-in function values, to determine whether the condition is enabled and whether an ON-unit is established, and then take the necessary action.

The methods used by the PL/I optimizing compiler are summarized below.

1. ### Detection of the PL/I conditions

   All PL/I conditions that correspond directly to program check interrupts are left to the detection of the operating system.

   A SPIE/ESPIE macro, issued during program initialization, results in control being passed to the error handling module IBMBERR.

   All other interrupts are detected by special checking-code, either generated by the compiler, or included in library modules. The checking-code calls the error handling module IBMBERR when a condition is detected.

2. ### Acquiring information about the interrupt

   Information about the interrupt is obtained by analyzing the PSW for program check interrupts and by checking-code for software detected interrupts. Condition built-in function values are accessed through a control block known as the ON communications area (ONCA).

   For software detected conditions, the ONCA is largely set up by the checking-code. For system detected conditions the ONCA is set up by the error handler from the information in the PSW.

3. ### Compilation and handling of ON-units

   Certain simple ON-units are represented by a series of flags in an ON control block (ONCB), but the majority are compiled as independent program blocks to which control is passed from the error handling module.

4. ### Maintaining a record of enablement and establishment

   During execution, information indicating which conditions are enabled and which ON-units are established is placed in the following control blocks:

   Enable cells
   : indicating enablement or disablement of the conditions that can be enabled and disabled by the programmer.

   ON-cells indicating which unqualified conditions have established ON-units.

   ON control blocks (ONCBs)
   : indicating address of ON-units or action to be taken, and for qualified conditions, whether the ON-unit is established, and, for CHECK only whether the condition is enabled.

5. ### Determining and directing action when interrupt occurs

   After every interrupt, control is passed to the error-handling module IBMBERR.

   A test is first made to see whether the condition is one that may be enabled or disabled by the programmer. If the

condition is disabled, control is returned to the point of
interrupt. If the condition is enabled, a search is made in
all active blocks for an established ON-unit. This is done
by examining ON-cells or ONCBs set up by compiled code. If
an ON-unit is found, the specified action is taken. If the
dummy DSA is reached without finding an ON-unit, standard
system action is taken under the control of the error
handling module.

The scheme is shown diagrammatically in Figure 46 and each topic
is discussed in greater detail in the following sections. A
summary of the uses of the various control blocks is given in
Figure 47 on page 113.

DETECTING CONDITIONS

1   SYSTEM   2   COMPILED CODE OR LIBRARY ROUTINES

```
Use system facilities           Execute checking code
if possible. SPIE/ESPIE         for all enabled conditions
macro, issued during            not detected by system.
initialization, passes          Call error handler when
control to error hand-          condition detected.
ling module when
interrupts occur
```
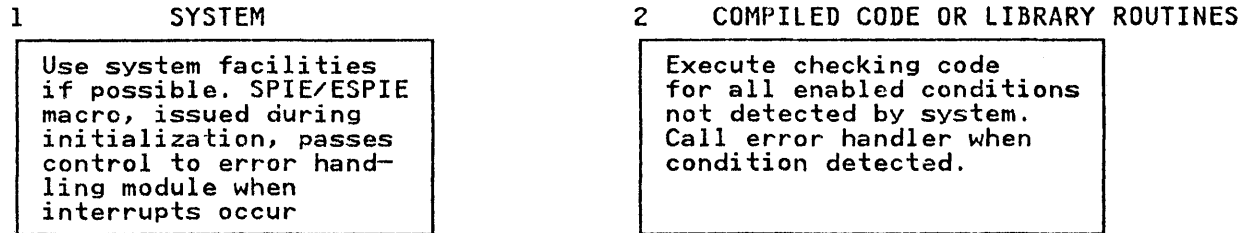
INDICATING ACTION REQUIRED WHEN CONDITION OCCURS
                                     |
COMPILED CODE   V

```
Set up flags indicating which
conditions are enabled.
Set up control blocks indicating
which ON statements have been
executed and, consequently,
which ON-units are established
and the addresses of such
ON-units.
```

CONTROLLING ACTION AFTER CONDITION HAS OCCURRED
                                   |
ERROR HANDLING MODULE — IBMBERR   V

```
From information set up in control blocks and flags by compiled code,

of the following actions to take when an interrupt has occurred
```

```
If condition disabled     If ON-unit established    If no-unit established

Ignore interrupt and      Take action specified     Take standard system
return.                   in ON-unit. If no         action as defined in
                          GOTO out of ON-unit       the language
                          take action specified
                          for normal return
```

Figure 46.   The Principles of Error Handling

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 word. (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 indicated 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 back-chain field.

Figure 47 (Part 1 of 2). The Major Fields Used in Error Handling

DSA

Flags	
Address of LWS	

Enable cells
Block	Current

Address of static ONCBs

Dynamic ONCB chain

Address of ON-cells

ON-cells
1st	2nd	3rd

End of chain (Zero)

Dynamic ONCB
Address of FCB
Code	Flags
Address of ON-unit

Back-chain

Address of symbol table

Dynamic ONCB
Code	Flags
Address of ON-unit

Back-chain

Address of FCB

Dynamic ONCB
Code	Flags
Offset of label variable
(ON-unit is GOTO only)

Address of label variable

LWS

Flags	ONCA offset
1st level	
Flags	ONCA offset
2nd level	
Back-chain to dummy ONCA	
Condition built-in function information	ONCA

STATIC storage

Code	Flags	1st static ONCB
Address of ON-unit		
Code	Flags	2nd static ONCB
Address of ON-unit		
Code	Flags	3rd static ONCB
Not set (system action		

Figure 47 (Part 2 of 2). The Major Fields Used in Error Handling

Figure 48 on page 115 gives a programming example in which the error handling actions can be followed through. Figure 47 on page 113 summarizes the complete error handling operation. It is intended for reference throughout the chapter and for use as a reminder by readers who know the basic principles.

```
+-----------------------------------------------------------------+
|                      SOURCE PROGRAM                             |
+-----------------------------------------------------------------+
| (SUBSCRIPTRANGE): SORT:                                         |
| PROCEDURE OPTIONS (MAIN);                                       |
| ON SUBSCRIPTRANGE BEGIN;                                        |
| PUT EDIT ('SUBSCRIPTRANGE OCCURRED')(A);                        |
| PUT SKIP DATA (I,J,K);                                          |
| /*SUBSCRIPT VALUES FOR TEST*/                                   |
| END;                                                           |
|    .                                                           |
|    .                                                           |
|    .                                                           |
| ON SUBSCRIPTRANGE SYSTEM;                                       |
|    .                                                           |
|    .                                                           |
| END SORT;                                                       |
+-----------------------------------------------------------------+
```

```
+-----------------------------------------------------------------+
|                  ACTION DURING COMPILATION                     |
+-----------------------------------------------------------------+
| 1. Remove the ON-unit from the position it holds in the        |
|    block and treat it as a separate begin block.               |
|                                                                 |
| 2. Generate code to set a flag in the block enable cell of the |
|    DSA, to indicate that SUBSCRIPTRANGE is enabled             |
|    throughout the block.                                       |
|                                                                 |
| 3. Generate code to set up two ON-cells in the DSA. Set up     |
|    two corresponding ONCBs in the static internal control      |
|    section (one for each ON statement in the block).           |
|                                                                 |
| 4. Place instructions equivalent to the ON statements in       |
|    compiled code. The first statement causes a code byte       |
|    corresponding to SUBSCRIPTRANGE to be inserted in           |
|    the first ON-cell; the second statement causes the same     |
|    code byte to be inserted in the second ON-cell, and sets    |
|    the first ON-cell to zero.                                  |
|                                                                 |
| 5. Generate code to insert flags in the ONCBs. Insert the      |
|    address of the ON-unit in the first  ONCB.                  |
|                                                                 |
| 6. Generate code to carry out the ON-unit.                     |
|                                                                 |
| 7. Generate code to check for the occurrence of SUB-           |
|    SCRIPTRANGE in every statement that could potentially        |
|    cause the condition to be raised.                           |
+-----------------------------------------------------------------+
```

Figure 48 (Part 1 of 2). An Example of Error Handling

```
+----------------------------------------------------------------------+
|                      ACTION DURING EXECUTION                         |
+----------------------------------------------------------------------+
| 1. The checking code generated by the compiler recognizes the        |
|    occurrence of SUBSCRIPTRANGE and passes control to the            |
|    error handler, after placing any required condition built-        |
|    in function values in the ONCA. (In this case only the            |
|    error code is generated.)                                         |
|                                                                      |
| 2. The error handler checks to see if SUBSCRIPTRANGE is one          |
|    of those conditions that can be enabled by the programmer.        |
|    Since it is such a condition, a check is made, in the             |
|    block enable cells of the DSA, to see if it is enabled.           |
|    (If it were not enabled, control would return directly to         |
|    the point of interrupt).                                          |
|                                                                      |
| 3. Finding that the condition is enabled, the error handler          |
|    then goes to the ON-cells in the DSA. These are tested,           |
|    using a translate-and-test table in the TCA, to see if            |
|    SUBSCRIPTRANGE is established. After this, the action             |
|    depends on whether the code for SUBSCRIPTRANGE is                 |
|    detected in the first or second ON-cell, and consequently         |
|    whether the first or second ONCB is used.                         |
|                                                                      |
| 4. If the first ONCB is used, ON-unit action is indicated; if        |
|    the second ONCB is used, standard system action must be           |
|    taken. (Standard system action would also be taken if the         |
|    code for SUBSCRIPTRANGE were not found in the DSA ON-cells        |
|    of the block in which the interrupt occurred, or in the DSA       |
|    of any dynamically encompassing block.)                           |
+----------------------------------------------------------------------+
```

```
+-------------------------------------+  +-------------------------------------+
| ON-unit action                      |  | System action                       |
|                                     |  |                                     |
| 1. A further allocation of library  |  | 1. For SUBSCRIPTRANGE, standard     |
|    workspace and a new ONCA are     |  |    system action is to produce a    |
|    acquired in case they should be  |  |    message and raise ERROR. The     |
|    needed during execution of the   |  |    message modules are called       |
|    ON-unit.                         |  |    to put out a message dependent   |
|                                     |  |    on the error code.               |
| 2. The ON-unit (addressed from the  |  |                                     |
|    ONCB) is executed.               |  | 2. ERROR is raised, and a search    |
|                                     |  |    is made through all active       |
| 3. Provided there is not a GOTO out |  |    blocks for an ERROR ON-unit.     |
|    of the ON-unit, return is made   |  |    Since there is none, standard    |
|    to the error handler. The error  |  |    system action is again taken;    |
|    handler the carries out standard |  |    this is to raise FINISH. Since   |
|    system action for return from an |  |    there is no FINISH ON-unit, the  |
|    ON-unit.                         |  |    standard system action of        |
|                                     |  |    returning to IBMBPIR is taken,   |
|                                     |  |    thus terminating the program.    |
+-------------------------------------+  +-------------------------------------+
```

Figure 48 (Part 2 of 2).   An Example of Error Handling

The handling of the CHECK condition, which is a special case, is
treated in a separate section of this chapter under the heading
of "The CHECK Condition."

DETECTING THE OCCURRENCE OF CONDITIONS

SYSTEM DETECTED CONDITIONS

As far as possible, the detection of PL/I conditions is left in the hands of the operating system. Those conditions that can be detected by the operating system are left in the hands of the operating system. The only interrupt that is masked out in the PSW is the significance exception. Regardless of the enablement or disablement of the PL/I conditions no other interrupts are inhibited.

When a condition is detected by the system, a SPIE/ESPIE macro, executed during program initialization, causes control to be passed to entry point A of the error-handling module IBMBERR. The address of this point is held in the TCA appendage. When entered by this entry point the error handler equates the interrupt with a PL/I condition and passes control to the main error handling logic of the module. The relationship between PL/I conditions and system interrupts is shown in Figure 43 on page 105.

SOFTWARE DETECTED CONDITIONS

During compilation, the compiler analyzes the conditions enabled for each block and statement. The analysis ensures that the necessary checking code is executed. The checking code may be specially generated by the compiler, or it may be included in library modules that will be called when the particular condition is enabled. The method used for checking each condition is shown in Figure 45 on page 108.

As far as possible the checking code is not included in the program if the condition that it checks for is not enabled. However, every library module contains the checking code for detecting any PL/I condition that can occur in the module. In certain circumstances, therefore, code to check software detected conditions will be executed and a call made to the error handler even though the condition is disabled.

When an interrupt has been detected during execution, the checking code sets up a parameter list for the error handling module IBMBERR. This parameter list, known as the interrupt control block contains a code that defines the type of interrupt that has occurred and, if the condition is qualified, contains a means of identifying the qualifier. The checking code also calculates the value of relevant built-in functions and places these values, or their addresses in a control block known as the ON communications area (ONCA).

When these actions have been carried out a call is made to entry point IBMBERRB, of the error handling module IBMBERR. The address of this entry point is held at the offset X'78' in the TCA.

Detecting I/O Conditions

The TRANSMIT and the ENDFILE conditions are normally detected by the data management routines rather than by PL/I code. When this occurs the error or end-of-file routine in the PL/I transmitter modules receives control and passes it to the error handler via a special I/O error module. This I/O error module contains the necessary code to set up the interrupt control block, including the error code and the qualifier. These conditions can, therefore, be considered to be software detected. Further detail is given in Chapter 8, "Record-Oriented Input/Output" on page 154.

EXECUTING SIGNAL STATEMENTS

SIGNAL statements take the same form as software detected interrupts, they are executed by a call to IBMBERR with the appropriate interrupt control block. The error code in the interrupt control block will indicate, to the error handler, the type of condition signalled, and the fact that the condition was signalled. The call to the error handler is made to the entry point B, regardless of whether the condition is normally detected by system or software.

It is necessary for the error handler to know that the condition was signalled, because different action may be required if the interrupt was signalled when computing certain built-in function values.

PASSING INFORMATION ABOUT INTERRUPTS

When the error handler is entered it must be able to access information about the interrupt. This information must identify the type of condition that has occurred and further identify the interrupt so that the most useful diagnostic message can be generated. Any relevant built-in function values must be available, plus the default values for any built-in functions that are not relevant to the type of interrupt.

When the interrupt is software detected, some of the information is set up in the checking code before control is passed to the error handler. When the interrupt is system detected, the PSW is used and the error handler interprets the information in the PSW, setting up information in a format similar to that produced by the checking code. This allows the main logic of the error handler to treat program checks and software detected conditions in the same manner.

When the error handler is entered in an MVS/Extended Architecture environment, it will switch the addressing mode (AMODE) to 31-bit addressing.

The parameters passed to the error handler by compiled code are known as the interrupt block, and take the following format:

Word 1   Error code

Word 2   Qualifier if any

Words 3, 4 and 5   Extra information used in handling CHECK

The error code defines the type of error. The qualifier gives a method of identifying the qualifier for qualified conditions. For unqualified errors, the interrupt block may be only 1 word. For I/O conditions, the address of the DCLCB is used as qualifier. The address of a symbol table, control section, or pseudo register offset is used for other qualified conditions.

The address of software detected interrupt is taken from the register 14 value when the error handler is called with a BALR 14, 15. This value is stored in the DSA by the prolog of the error handler. When the interrupt is system detected the address is taken from the PSW. See Chapter 12, "Debugging Using Dumps" on page 248, for a discussion of the register usage during hardware and software interrupts.

ERROR CODE

The error code is either a two or four byte code that defines the reason for the interrupt. For all conditions except the error condition a four byte code is passed. For the errors that will immediately raise the ERROR condition only a two byte code is passed.

The 4-byte code is as follows:

Byte 1   identifies the PL/I condition

Byte 2   identifies the cause of the condition

Byte 3 and 4   identify those ON built-in functions that are valid for the condition.

The two byte error code is raised only for the ERROR condition. The ERROR condition is raised for those interrupts and errors that have no directly associated PL/I condition. Certain of these, such as taking the square root of a real negative number, are software detected. Others are associated with program checks interrupts, such as a data interrupt.

When the error condition is to be raised a two byte code only is generated. The value in this code corresponds with a table held in the error handler which identifies the cause of the interrupt. Error codes are listed in Figure 105 on page 256.

CONDITION BUILT-IN FUNCTIONS

Certain condition built-in function values are implicit in the information that is passed to the error handler. ONCODE, for example, bears a direct relationship to the error code. Other values, such as ONCHAR and ONSOURCE must be calculated when the interrupt occurs. These values or the addresses of the values are placed in the ONCA. The ONCA is addressed from library workspace. The address of library workspace is held at a fixed offset in every DSA. ONCODE, ONLOC, and ONFILE are not generated by the checking code as their contents are implicit in the information passed to the error handler.

The ONCODE is deduced from the error code and, when required, a transient library module IBMBEOC is called to translate the error code into the ONCODE. Both an error code and an ONCODE are used as it is possible to define the error more accurately than can be done with the ONCODEs, which must be kept compatible with other PL/I compilers. Thus the error code allows a more useful diagnostic message to be generated than would be possible if only the ONCODE was generated.

The ONLOC value is also calculated by a separate module. ONFILE is accessed from the DCLCB. Both ONLOC and ONFILE are placed in the ONCA only if an ON-unit is to be entered. Similarly if an ON-unit is to be entered the error code is placed in the ONCODE field of the DSA. If the ONCODE value is required in the ON-unit the module IBMBEOC is called to calculate the ONCODE from the error code.

Chain of ONCAs

PL/I allows access to condition built-in function values when no condition has occurred or when a condition has occurred in which the built-in function is invalid. The rule is, the built-in function value given is the most recent value in an active ONCA or the default value. To allow for this, ONCAs are chained together and at the end of the chain is the dummy ONCA that is set up in the program management area during the program initialization. The dummy has the same format as the other ONCAs and contains the default values or pointers to the default values for all built-in functions.

For every interrupt that occurs, a new ONCA is acquired. This means that, should a condition occur within an ON-unit, an ONCA will be available in which to place any relevant built-in function value or their addresses. A new allocation of library workspace (LWS) is also required for use during the ON-unit.

When a built-in function is required, the ONCA before the current ONCA is inspected. The current ONCA is unused as it is ready for a new set of values. Each ONCA is headed by flags that indicate which built-in functions are given in the ONCA. When the required built-in function value is flagged as invalid, a chain back is made to the previous ONCA. As all fields are valid to the dummy, the default will be used if there have been no interrupts for which the function is valid.

In the program below, an example of the chain of ONCAs is shown. The ONCHAR reference in the NAME ON-unit would be valid if the NAME condition was raised in the CONVERSION ON-unit. The correct value would be accessed after chaining back to the ONCA associated with the CONVERSION interrupt.

In other circumstances the default value would be accessed from the dummy ONCA.

```
CHAIN:    PROC(OPTIONS(MAIN));

          ON NAME BEGIN;        /*NAME ON-UNIT*/
          PUT DATA(ONCHAR);
          •
          GOTO LABEL1;
          END;

          ON CONVERSION BEGIN;  /*CONVERSION ON-UNIT*/
          •
          GET DATA (A,B,C);
          •
          •
          END;

LABEL1:   X=Y=2
          •
          END CHAIN;
```

A situation that could occur in this program and the associated chaining of ONCAs are shown in Figure 49 on page 121.

When an ON-unit is completed, the latest generation of LWS and ONCA are deleted; control returns to a block before the error handler. This is because they are held as VDAs associated with the error handler's DSA. When control leaves the error handler, the current ONCA will contain the interrupt information for the original interrupt. This information remains until the ONCA is freed or a further interrupt occurs, in which case it is overwritten. (See Figure 49 on page 121.)

Figure 49. Accessing a Built-In Function Value from the Chain of ONCAs

PROGRAM FLOW

MAINLINE CODE

CONVERSION occurs ONCHAR & ONSOURCE addresses placed in 1st ONCA Error code placed in ONCODE field

Continuation of mainline code

CONVERSION on-unit entered. New ONCA acquired

CONVERSION ON-UNIT

NAME condition occurs. DATAFIELD address placed in second ONCA. Error code placed in ONCODE field.

Continuation of on-unit

NAME on-unit entered. New ONCA acquired.

NAME ON-UNIT

GOTO causes return to mainline code

MAINLINE CODE

CHAIN OF ONCAs

Dummy ONCA
Holds all default values or their addresses

1st. ONCA
Ready for use

Dummy ONCA
Holds all default values or their addresses

1st. ONCA
ONCODE, ONCHAR, & ONSOURCE fields filled in

2nd ONCA
Ready for use

Dummy ONCA
Holds all default values or their addresses

1st. ONCA
ONCODE, ONCHAR, & ONSOURCE fields filled in.

2nd. ONCA
ONCODE & DATAFIELD filled in.

3rd. ONCA
Ready for use.

Dummy ONCA
Holds all default values or their addresses

1st. ONCA
ONCODE, ONCHAR, & ONSOURCE fields unaltered. Will be overwritten if there is a further interrupt

---

All condition built in functions accessed from dummy ONCA

ONCODE, ONCHAR, and ONSOURCE accessed from first ONCA. All others from dummy ONCA

ONCODE and DATAFIELD accessed from second ONCA. ONCHAR and ONSOURCE accessed from first ONCA. All others from dummy ONCA.

On return from an on-unit all unnecessary ONCAs are discarded as they are in the LIFO stack. The current ONCA retains the previous interrupt information until over-written. Values are taken from the dummy.

ESTABLISHMENT AND ENABLEMENT INFORMATION

(Executing ON Statements)

Establishment and enablement information is set up and updated by compiled code. Enablement is indicated by a set of flags known as the "current enable cells" which are held in every compiled code DSA. Establishment for unqualified conditions is indicated by a further series of bytes in the DSA known as the ON-cells. Establishment for qualified condition is indicated in flags in dynamic ONCBs. Dynamic ONCBs are held in the DSA of the block in which the associated ON statement occurs.

To alter the enablement for the duration of a statement or to execute an ON statement, compiled code alters the appropriate fields mentioned above.

In an MVS/Extended Architecture environment, an additional test is made. ON-units for programs that are running in 24-bit addressing mode are ignored for interrupts that occur in 31-bit addressing mode.

ENABLEMENT

Enablement is indicated in the current enable cells, a two byte field held at offset X'56' in the DSA. Each condition whose enablement is under programmer control has a bit allocated to it. The conditions associated with each bit are shown in Figure 50.

```
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       STRINGNAMES

Bit 9       STRINGRANGE

Bit 10      CHECK*

Bit 11      CHECK*

Bits set to 0 if condition enabled

* See section "The CHECK Condition for details"
```

Figure 50. Meaning of Enablement Bits

The CHECK condition has three bits associated with it. This is because the CHECK condition can be used both as a qualified and as an unqualified condition. Bit zero indicates that CHECK is enabled, either qualified for one or more variables, or unqualified for all variables. Bit 11 indicates that CHECK has been enabled or disabled as an unqualified condition. Bit 10, only valid if bit 11 is set, indicates whether the unqualified CHECK is enabled or disabled. The CHECK condition is further

described in "Raising the CHECK Condition" on page 131 and illustrated in Figure 53 on page 133.

A further two byte field in the DSA held at offset X'54' is known as the block enable cells. This field is similar to the current enable cells and holds a record of the enablement at the start of the block.

Both current enable and block cells are set up by the prolog code. If the enablement is altered for the duration of a statement, the appropriate bit in the current enable is altered at the start of the statement. At the end of the statement the bit is reset to its previous value. If there is an interrupt during the execution of the statement, ON-unit action may return control to another part of the block where different conditions are enabled. The block enablement cells are necessary to allow for this. Whenever a GOTO out-of-block occurs in an ON-unit the GOTO code in the TCA resets the current enable cells from the block enable cells. This ensures enablement will be correct, regardless of the situation when control left the block.

Qualified Conditions

The only qualified condition whose enablement is under programmer control is the CHECK condition. As CHECK is a special case it is treated in detail elsewhere. The principle involved however is that enablement for any particular qualifier is given in a dynamic ONCB and, to discover whether a CHECK is enabled for a particular item, a search must be made in the DSA chain for a relevant dynamic ONCB.

ESTABLISHING AND EXECUTING ON AND REVERT STATEMENTS

For establishment the situation differs between qualified and unqualified conditions. This is because at any one point in the program there can only be one established ON-unit for an unqualified condition but there can be an unlimited number of established ON-units for qualified conditions. In a program with a number of files, for example, the programmer may wish to take different action when the end of the data is reached in each of the files. Consequently, there could be an established ENDFILE ON-unit for each file.

ON-units are established by the execution of an ON statement. Once it has been discovered that an ON-unit is established it is then necessary to access the ON-unit. Access to the address is made through a control block is known as the ON-control block ONCB. For unqualified conditions, ONCBs are set up during compilation in static internal storage and are known as static ONCBs. For qualified conditions, ONCBs are set up (by compiled code) in the DSA and are known as dynamic ONCBs. See Figure 51 on page 124.

DSA of block

Object module

```
Standard
House-
keeping
area of
DSA
```

A(1st. Static ONCB) (Offset X'5C')	→	1st. Static ONCB Contains address of ON-unit and other information
A(Latest Dynamic ONCB) (Offset X'60')		2nd. Static ONCB Contains as above
		3rd. Static ONCB Contains as above
A(ON-cells) – Offset X'70'		
ON-cells [X]		ON-unit
End Marker ON-cells Contain Code for Condition if ON-unit is Established; Otherwise 0		ON-unit
Dynamic ONCB Contains Code Indicating Condition Qualifier and Whether ON-units are Established. Also Address ON-unit.		ON-unit
Further Dynamic ONCB Contents as above. Dynamic ONCBs are chained.		ON-unit
		ON-unit

Key

— — — — — Broken lines show method of addressing ON-units for qualified conditions. The ONCBs are chained together and the address of the end of the chain held at a fixed offset in the DSA. The ON-unit (if any) is addressed from the ONCB.

─────────── Solid line shows method of addressing ON-unit for unqualified condition, ONCBs are held contiguously in the same order as ON-cells, and the address of the first ONCB is held at a fixed offset in the DSA. By determining the position of the relevant ON-cell, the position of the required ONCB can be inferred and hence its offset from the start of the static ONCBs. The first ON-cell refers to the first ONCB etc. The ON-unit is addressed from the ONCB.

Figure 51. Addressing ON-Units

Qualified Conditions

The establishment of qualified conditions is indicated directly in the ONCB. All dynamic ONCBs for a block are chained together and the address of first ONCB on the chain is held in a field at offset X'60' in the DSA. (See Figure 49 on page 121.)

Dynamic ONCBs contain a code indicating the condition type, flags to indicate whether the condition is enabled and whether the associated ON-unit is established, a method of identifying the qualifier, and, either the address of the compiled code ON-unit, or flags indicating the action specified in the source program ON-unit. There is an ONCB for every ON statement in the block that refers to a qualified condition.

ON AND REVERT STATEMENTS: When the ON statement is executed the appropriate dynamic ONCB is set up, chained, and the establishment bit in the ONCB is set "on" by the compiled code. For second and subsequent ON statements or REVERT statements for the same condition and qualifier, the information in the ONCB (flags and address of ON-unit) is altered.

Unqualified Conditions

For unqualified conditions establishment information is held in a series of one byte fields known as ON-cells. There is one cell for each ON statement in the block and, consequently, for each ONCB associated with the block. ONCBs for unqualified conditions are held contiguously in static internal storage in program block order. (See Figure 49 on page 121.)

In each DSA containing ON statements an area is reserved for ON-cells. Cells are one byte fields that correspond one-for-one with the static ONCBs for that block. The first ONCB for the block is addressed from offset X'5C' in the DSA. On cells are initialized to zero by the prolog code. When the ON statement associated with the ON-unit is executed, a code is set in the ON-cell indicating the condition type. The error handling module searches for an established ON-unit by testing the ON-cells in the DSA of each active block until, either an active ON-cell for the condition is found, or the major task dummy DSA is reached. When an active ON-cell is found, the number of ON-cells in the block preceding the active ON-cells are calculated. The associated static ONCB will be in the same relative position. As all ONCBs for unqualified conditions are the same length the address of the requested ONCB can be determined and the action to be taken decided from the ONCB.

ON AND REVERT STATEMENT: When an ON statement is executed a code indicating the condition type is set in the appropriate ON-cell. If there was a previous ON statement for the condition the former ON-cell is set to zero. For REVERT statements any ON-cell referring to the condition is set to zero.

If there is more than one ON statement for the same condition in a block, the flags in the previous ON-cell will be set off 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.

HANDLING ON-UNITS

ON-units, except certain single statement ON-units, are treated as separate program blocks by the compiler. They are separated from the ON statement and compiled with prolog and epilog code. The address of the ON-unit is placed in an address constant. The ON statement remains in its logical place in the program and sets either the ON-cells or a flag in the dynamic ONCB, to indicate that the associated ON-unit is established.

In order to save the overhead of executing prolog and epilog code, certain single-statement ON-units are not compiled. Instead the action required is indicated by flags in the ONCB and is carried out under the control of the error handling module.

The types of ON-unit involved are:

1. Null ON-units

2. ON-units containing only SNAP, SNAP SYSTEM, OR SYSTEM options.

3. ON-units containing only a GOTO statement.

The presence of these ON-units is indicated by flags in the associated ONCB. For the GOTO only ON-unit, the ONCB also contains the offset in the DSA of the label variable or label temporary to which the GOTO is to be made.

THE LOGIC OF THE ERROR HANDLER

A simplified flowchart of the error handling module IBMBERR is given in Figure 52 on page 127. This flowchart shows the action during the handling of an interrupt and includes the execution of an ON-unit. The logic is described below. A complete description is given in the licensed program product document OS PL/I Resident Library Program Logic.

IBMBERR—ERROR HANDLING MODULE

The error-handling module, IBMBERR, handles three situations. These are:

1. Program check interrupts.

2. PL/I conditions detected by the object program.

3. Errors detected by the object program that are not directly related to the 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. When 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.

Software interrupts

```
  ( ENTRY POINT B )
          |
          v
┌──────────────────┐
│ Alter address-    │
│ entered-after-    │
│ hardware-         │
│ interrupts, to    │
│ entry point C     │
└──────────────────┘
```

System detected interrupts

```
  ( ENTRY POINT A )
          |
          v
┌──────────────────┐
│ Alter address-    │
│ entered-after-    │
│ hardware-         │
│ interrupts, to    │
│ entry point C     │
└──────────────────┘
          |
          v
┌──────────────────┐
│ Determine         │
│ condition type    │
│ from PSW          │
└──────────────────┘
```

Hardware interrupts during error handling

```
  ( ENTRY POINT C )
          |
          v
┌──────────────────┐
│ Call IBMBPEP      │
│ to put out        │
│ message           │
└──────────────────┘
          |
          v
┌──────────────────┐
│ Terminate         │
│ program with      │
│ ABEND macro       │
└──────────────────┘
```

┌──────────────────┐
│ Restore hardware │
│ interrupt address │
│ and return   │
└──────────────────┘

↑ YES

Is condition disabled? — YES — Can condition be disabled?

NO   NO

┌──────────────────┐
│ Call message   │
│ module to   │ — YES — Is it ERROR?
│ generate message │
└──────────────────┘

NO

Established ON-unit for condition in DSA? — YES — Special* ON-unit — NO —

┌──────────────────────┐
│ Acquire LWS and   │
│ ONCA. Restore   │
│ hardware interrupt   │
│ address to entry   │
│ point A   │
└──────────────────────┘

NO   YES

┌──────────────────┐   ┌──────────────────┐
│ Chainback to   │   │ Carry out   │
│ previous DSA   │   │ specified action │
└──────────────────┘   └──────────────────┘

┌──────────────────┐
│ ON-Unit   │
├──────────────────┤
│ Execute ON-unit   │
└──────────────────┘

NO — Reached major task dummy DSA?

┌──────────────┐
│ GOTO   │
│ out of   │
│ ON-unit   │
│ can occur   │
│ here   │
└──────────────┘

YES

┌──────────────────┐
│ Take standard   │
│ system action   │
└──────────────────┘

┌──────────────────┐
│ Reset hardware   │
│ interrupt address │
│ to entry point C │
└──────────────────┘

┌──────────────────┐
│ Take action for   │
│ normal return   │
│ from ON-unit   │
└──────────────────┘

```
/‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾\
│ Standard system action and action for normal return takes one of the forms │
│ shown below depending on the condition that caused the interrupt and        │
\ whether the interrupt was signalled.  Messages are generated for some conditions. /
 _____/
```

┌──────────────────┐   ┌──────────────────┐   ┌──────────────────┐
│ Raise further   │   │ Return to   │   │ Reset error   │
│ condition   │   │ IBMBPIR to   │   │ handling address │
│ Start ON-unit   │   │ terminate task   │   │ and return to   │
│ search in DSA   │   │   │   │ point of interrupt│
│ of interrupt   │   │   │   │   │
└──────────────────┘   └──────────────────┘   └──────────────────┘

* Special ON-units are not entered these are: null ON-units, or ON-units containing only a SNAP or SNAP SYSTEM instruction.

Figure 52.   Simplified Flowchart for IBMBERR

PROGRAM CHECKS INTERRUPT

Before a program check interrupt can be handled as a PL/I condition, action must be taken to prevent the system terminating the job should a further program check interrupt occur.

This is done by altering the old program PSW and returning out of the SPIE/ESPIE exit code so that it appears to the system that the interrupt has already been handled. The second word of the PSW passed to ERR in the PIE (program interrupt element) or the EPIE (extended program interrupt element) containing the interrupt address is stored in the register 15 field in the save area which was current when the interrupt occurred. IBMBERR then changes the address in the PSW in the PIE to an address in IBMBERR. Control then passes via the supervisor to the address in IBMBERR that has been inserted in the PSW. Handling of the interrupt consequently appears to the supervisor to be finished. The address, in the field of 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 IBMBERR, control will pass to IBMBERRC, which terminates the job.

The 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 IBMBERR'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 routine 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 overflow, exponent overflow, decimal overflow, or fixed-point divide has occurred, then it may correspond to the PL/I condition SIZE and not to FIXEDOVERFLOW or ZERODIVIDE. If this is possible, a flag will have been set in the program check interrupt qualifier in the TCA. A test of this flag is therefore made and the necessary action taken, SIZE being raised if it is enabled.

3. If the interrupt was an <u>operation interrupt</u> it may have been caused by an extended floating point instruction being used on a machine that does not have the extended float instruction set. If this is the case, the instruction may require simulation. The error handler therefore passes control to a module IBMBEEF that interfaces with the extended float simulator IEXPSIM. IBMBEEF passes control to the extended float simulator which returns the correct result if the statement was valid, or a return code if the statement was invalid. If the statement is valid, IBMBEEF returns control to the point of interrupt. If the statement is invalid, IBMBEEF returns control to the error handler.

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 4-byte code is passed. For other errors, the code consists of only two bytes. For the 2-byte code, the first byte indicates which class of error has occurred. For the 4-byte code, 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 ERROR or another PL/I condition. If the code indicates ERROR, then the message module IBMBESM is linked. This module prints the relevant diagnostic message: a suitable 4-byte code is then generated. The situation is then treated as for any other PL/I condition.

The second 2 bytes of code passed when a PL/I condition has been raised indicate which built-in functions are relevant to the condition. If the condition is one that needs to be qualified, the qualification is also passed.

When a PL/I condition error code is passed, action depends on whether the condition is one of those that can be 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. To find established ON-units, a test is first made in the action byte to discover whether the condition is qualified. If the condition is not qualified, 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 IBMBERR. This is done with a translate and test instruction using the TRT table addressed from the offset X'1C' in the TCA. When found, the position of the located ON-cell gives the position of the associated ONCB. A test can then be made to determine the action to be taken.

If the condition is qualified, a search for an active matching ONCB is carried out through the chain of dynamic ONCBs held in the DSA.

If the major task dummy DSA is reached without a match being found, then standard system action is taken. This action is defined in IBMBERR. When a matching active ONCB is found, tests are then made, as follows, on the flags in the ONCB.

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

Test 2   Is SYSTEM specified? (This can occur when "ON condition SYSTEM" was specified. If system is specified, then the action in IBMBERR is taken.

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

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

Test 5. Is this machine running under MVS/Extended Architecture? If so, ignore ON-unit established for programs running in 24-bit addressing mode if the interrupt occurs in 31-bit addressing mode.

If none of these 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 must be initialized and its address put into the standard offset in the DSA of IBMBERR. 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 address in the TCA, which was altered by the error handler, must also be restored to its original setting 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 IBMBERR 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. Four PL/I conditions cause action other than return to be taken.

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

FINISH
If the FINISH condition is raised then a return code is set in the correct field of the TCA, and GOTO performed to the termination routine IBMBPIR. (If finish is signalled, then return is made to the point of interrupt.)

CONVERSION
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.

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

RETURN TO THE 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.

Program Check Interrupts

For program check interrupts, the status of the program at the original point of interrupt has to be restored before return to the point of interrupt can be made. 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 DSA at initial entry to IBMBERR.)

In a non-MVS/Extended Architecture environment, the address in the TCA/PICA is altered so that the branch address, after a program check interrupt, is changed from IBMBERRC to another point in IBMBERR. An interrupt is then caused, and the supervisor gains control. Consequently, the address in IBMBERR 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. The point of entry for program check interrupts is then reset to IBMBERRA. Return is made to the address in the PSW, which is that of the original interrupt.

In an MVS/Extended Architecture environment, a separate ESPIE is issued and used. The same steps are followed as in a non-MVS/Extended architecture environment. Registers are copied back to the EPIE, the PSW is reset, and control returns to the supervisor.

THE CHECK CONDITION

The CHECK condition has to be handled in a different manner than 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 <u>unqualified</u>. CHECK or NOCHECK with a name list are referred to as <u>qualified</u>.

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 descendent, 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. However, for GET DATA statements there is no means of knowing which items will be passed in the data stream, and if the CHECK condition is enabled for any variable that could be read in, it is necessary to check every variable in the input stream to see whether CHECK is enabled for that variable. Consequently, when a GET DATA instruction is being executed, it is necessary for the error handler to test to see if the CHECK condition is enabled.

With the exception of the CHECK condition, all conditions whose enablement is under programmer control are <u>unqualified</u>. 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 qualified CHECK, however, there are many possibilities, 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 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 the following tasks.

1. Test to see if CHECK occurred during the execution of a GET DATA statement. If so, tests for enablement must be made. If not, continue with step 3.

2. Test to see if CHECK is enabled. This involves a search along the static back-chain to determine, for each block, first, if qualified CHECK is enabled or disabled for the particular name for which CHECK was raised, then, if unqualified CHECK is enabled or disabled.

3. Search for a qualified established ON-unit. This involves searching the dynamic back-chain for a relevant dynamic ONCB.

4. If there is no qualified established ON-unit search for an unqualified established ON-unit. This involves a further search of the dynamic back-chain looking for appropriate on-cells.

5. If no established ON-unit is found, take standard system action.

This process is illustrated in Figure 53 on page 133.

Testing for Enablement

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

Bit 0

'0'B   CHECK is enabled for certain items in this statement

'1'B   CHECK is disabled for this statement

Bit 10 (only valid if bit 11 is set)

'0'B   The unqualified prefix that applies is NOCHECK

'1'B   The unqualified prefix that applies is CHECK

Bit 11

'0'B   No unqualified prefix applies to this statement

'1'B   An unqualified prefix applies to this statement

Throughout this discussion, 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.

Start

Return

Return

Test 'any-check' enablement bit in current enable cell (bit 0)

Bit set on? — No → Return

Bit set on? — Yes → Search dynamic ONCBs for relevant qualified CHECK

Found? — Yes → Disabled?

Found? — No → Test further enablement bits (10 & 11) for unqualified CHECK

Disabled? — Yes → Return

Disabled? — No → Return to DSA of block in which CHECK was raised

Dummy DSA? — No

Dummy DSA? — Yes → Return

Chain back to previous DSA on static chain

Unqualified NOCHECK found? — Yes → Return

Unqualified NOCHECK found? — No → Chain back to previous DSA on static chain

Unqualified CHECK found? — No → Unqualified NOCHECK found?

Unqualified CHECK found? — Yes → Return to DSA of block in which CHECK was raised

Return to DSA of block in which CHECK was raised

Return to DSA of block in which CHECK was raised

Dummy DSA? — Yes → Return to DSA of block in which CHECK was raised

Dummy DSA? — No → Search dynamic ONCBs for qualified ON-statement

Search dynamic ONCBs for qualified ON-statement

Take standard system action

Chain back to previous DSA on dynamic chain

Qualified match found? — No → Chain back to previous DSA on dynamic chain

Qualified match found? — Yes → Any action specified?

Search on-cells for unqualified CHECK

Reached dummy DSA? — Yes → Take standard system action

Reached dummy DSA? — No → Search on-cells for unqualified CHECK

Enablement only specified

Any action specified? — No → Enablement only specified

Any action specified? — Yes → Take action specified in ONCB

Take action specified in ONCB

Unqualified CHECK found? — No → Chain back to previous DSA on dynamic chain

Chain back to previous DSA on dynamic chain

Unqualified CHECK found? — Yes → Take action specified in ONCB

Take action specified in ONCB

Figure 53.   Handling the CHECK Condition

The test for enablement begins by a test on the any-CHECK bit in the enable cell. If this is set to zero, 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 no such ONCB is found, the unqualified CHECK enablement bits are tested for unqualified enablement or disablement. 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.

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 back-chain is searched for relevant ONCBs. If one is not found, a search is made through the back-chain for enable cells that indicate unqualified CHECK. If nothing is found, standard system action is taken.

Standard System Action

Standard system action for CHECK is taken under the control of a special module IBMBERC. This module acquires the necessary symbol table address or addresses, places them in a VDA and passes control to the stream I/O initializing routine and, on return, to the data directed director module IBMBSDO. On completion of the operation IBMBERC returns control to IBMBERR.

ERROR MESSAGES

The library module IBMBESM is called by the error handler to transmit the system messages and find the on-code value by calling the ONCODE routine IBMBEOC; control is then passed to IBMBESN to finish the system message, or to go to generate the SNAP message if required. The text for the messages is taken from a series of message text modules. The particular message text module required and the message within the module are determined from the error code.

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:

```
IBM4021 'ONCODE'=3100 'FIXEDOVERFLOW'
CONDITION RAISED IN DECIMAL DIVIDE IN
STATEMENT 31 AT OFFSET 000A35 IN
PROCEDURES 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
STATEMENTxxx] (AT NEAR) OFFSET xxx IN
PROCEDURE xxx
```

and continues as for a SNAP SYSTEM message.

The statement number is not always present in messages as 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 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 at a fixed offset from each entry point 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 and COUNT Options" on page 141).

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

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.

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 IBMBERR was called.

If the interrupt was 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 IBMBERR 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.

Identifying Entry Point Name and Statement Number

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 address of the entry point used before the interrupt is held in the save area of this DSA as the branch register contents. The dummy DSA ensures that a back-chain can be made from the main procedure DSA.

The name of the entry point is found by chaining back one DSA beyond the first procedure-DSA reached. This DSA holds the

address of the procedure-DSA entry point in the register 14 slot of its register save area (offset X'10' from the head of the DSA). The length of the name is held in a 1-byte field immediately preceding the entry point. The name immediately precedes the length field.

Statement numbers are generated separately for each external procedure, and the statement number table holds offsets from the first entry point in the external procedure.

When the statement number table is link-edited, the address of this entry point is placed at the head of the table. Consequently, the required offset can be found by comparing the address of the statement causing the error with the address of the first entry point held in the statement number table.

If the NUMBER option is in force, the numbers are held in 4-byte form preceded by a halfword statement number. Otherwise, the statement numbers are held in 2-byte form. Flags indicating which options are in use are held in the DSA. The DSA is further described in "Dynamic Storage Area (DSA)" on page 346.

As the offsets may be up to 6 bytes in length, a device is used for statement numbering whereby the table is divided into sections that correspond to the offset values that are held in the first 2 bytes of the offsets. Thus offsets starting X'00' are held in the first section of the table, offsets starting X'01' in the second, and so on. Each section of the table is headed by a pointer to the start of the following section, or set to zero if there is no following section. The complete table is also headed by the value of the maximum offset, so that offsets beyond the program can be readily detected.

The statement number is found by searching the correct section of the table for the first offset that is less than or equal to the last 4 hexadecimal digits of the calculated offset.

For SNAP messages, once the ON-unit has been found and the appropriate message generated, the rest of the trace gives information about procedures, begin blocks and ON-units. Thus all compiled code DSAs can be treated in the same way.

Filename and Name of CONDITION Condition

If the error was in I/O, then the address of the DCLCB of the file is passed to IBMBERR which stores it for IBMBESN to find the file name. Similarly, the address of the control section containing the condition name is passed to IBMBERR if the CONDITION condition is raised, and IBMBESN puts out the required section of message.

MESSAGE TEXT MODULES

The message module IBMBESM 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 the message text modules. The message required is determined from information in the error code. IBMBESN 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.

DIAGNOSTIC FILE BLOCK

Every attempt is made to put out error messages on the standard print file SYSPRINT. However, there are no reserved words in PL/I and consequently the name "SYSPRINT" may be used for a file with attributes other than PRINT OUTPUT, or may be use for a variable of any other data type. If SYSPRINT is declared as an unsuitable type of file it cannot be used for error messages and all error messages are written on the console.

A control block, the diagnostic file block (DFB), is set up during program initialization to indicate whether SYSPRINT can be used for error messages. If SYSPRINT has been declared as a file the address of the DCLCB is placed in the DFB. The DFB (diagnostic file block) is addressed from the TCA. When an error message module is to be put out, IBMBESM or IBMBPEQ inspects the DFB to see if SYSPRINT can be used for the message. If the flags in the DFB indicate that SYSPRINT cannot be used, the module IBMBEDO is called.

IBMBEDO tests to see if SYSPRINT is open. If it is not, IBMBEDO calls IBMBOCL to open it with the attributes STREAM PRINT. If SYSPRINT has been declared as a file the address of the DCLCB is picked up from the DFB. Should the attributes STREAM AND PRINT be incompatible with the declared or default attributes this is diagnosed by the OPEN module and appropriate flags are sent in the DFB to indicate that SYSPRINT cannot be used for error messages. This action does NOT raise the error condition.

If SYSPRINT has not been declared, a DCLCB will be generated and SYSPRINT will be opened, provided that the error occurs before a task has been attached. if a task has already been attached, or if the error occurs in an attached task, then SYSPRINT cannot be opened and all error messages are passed to the console.

If SYSPRINT is already open with unsuitable attributes this will have been flagged in the DFB and the messages will again be passed to the console.

If SYSPRINT has been declared as a data type other than a file this is flagged in the DFB and the error messages are set to the console.

If SYSPRINT has not been declared at all, a diagnostic SYSPRINT is opened and used, provided that there is a DD card for SYSPRINT.

DUMP ROUTINES

A series of library modules are provided to implement the PLIDUMP facility. Module IBMBKDM is the dump bootstrap module which is part of the resident library. This loads and calls the transient dump control module IBMBKMR, which in turn links and calls those modules required to carry out the dump options specified in the call to PLIDUMP. Several transient modules are used to reduce the amount of storage used at any one time. The organization of these modules is shown in Figure 54 on page 138.

In order to ensure that as much information as possible is provided when a call to PLIDUMP is made, a special SPIE/ESPIE macro instruction is issued at the start of every transient routine to intercept program check interrupts during the routine. When a program check interrupt occurs, an attempt is made to continue with the dump. 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, the "H" option is assumed and a hexadecimal dump produced. If the interrupt occurs during the execution of the hexadecimal dump module, a SNAP macro instruction is issued by the dump control module and a SNAP dump is completed under the control of the supervisor. When the SNAP

dump is completed control returns to the dump control module and the PLIDUMP is completed as requested in the dump options.

As further insurance against error, the dump control module IBMBKMR is divided into sections, and if an interrupt occurs in any of these sections, control is passed to a predefined address at the end of the section. Processing then continues from that point.

The dump modules are fully described in the publication OS PL/I Transient Library Program Logic.

CALL PLIDUMP   COMPILED
                                                CODE

```
                    +------------------------+
                    |       IBMBKDM          |      RESIDENT
                    +------------------------+      MODULE
                    |   Bootstrap module     |
                    +------------------------+
                              V   Link
                            --A--
+------------------+  LINK   +------------------+   LINK  +------------------+
|    IBMBPES       | <--->   |     IBMBKMR      |  <--->  |     IBMBKDO      |
+------------------+         +------------------+         +------------------+
|  ABEND Analyzer  |     +<--|  Dump            |         | Open Dump        |
+------------------+     |   |  Control Module  |         | File Module      |
                         A   +------------------+         +------------------+
                         LINK   LINK    V  LINK               LOAD
  +------------------+         +------------------+         +------------------+
  |    IBMBKPT       |         |     IBMBKFA      |         |   IBMBKDT/B      |
  +------------------+         +------------------+  >      +------------------+
  | Dump Parameter   |         | File Attributes  |-<--->  | Dump File        |
  | Translate Module |      V  | Dump Module      |  >     | Transmitter      |
  +------------------+         +------------------+  >     +------------------+

  +------------------+         +------------------+         TRANSIENT
  |    IBMBEOC       |         |     IBMBKTC      |
  +------------------+         +------------------+         MODULES
  | ON-Code          |         | Trace check      |
  | Calculator       |         | module           |
  +------------------+         +------------------+
  V LOAD                   V
  A                        XCTL
  +------------------+
  |    IBMBKTR       |
  +------------------+
  | Dump trace       |----->        <----
  | analyze          |
  +------------------+
  V  XCTL                                     A
  +------------------+
  |    IBMBKTB       |
  +------------------+
  |  Control         |
  +------------------+
```

Figure 54.   Interrelationship of Dump Routines

In order to avoid mixing of PL/I dump and other information, dump data is not transmitted to any PL/I file. A special dump file known as PLIDUMP is used for the output of the dump modules. This file has its own transmitter and a special opening module IBMBKDO. A control block, the dump block, (DUB) is set up during program initialization and is used to hold information about the status of the dump file and to simplify access to the file. The DUB (dump block) is addressed from offset X'20' in the TCA appendage. To generate a PL/I dump, it is necessary to have a DD card for PLIDUMP, or PL1DUMP.

Before any output has been produced by the dump modules, the dump control module IBMBKMR inspects the DUB to see if the dump file is open. If the dump file is not open, and is not flagged as unopenable, the control module calls the dump file open routine (IBMBKDO) to open the file. IBMBKDO acquires space for the necessary control blocks, loads the dump transmitter and attempts to open the dump file.

If the attempt to open the dump file fails, IBMBKDO flags the DUB and returns. The DUB flags are tested by IBMBKMR, and if the file has not opened, a message is put out and the dump is terminated. The job is either continued, terminated or an exit is made from the task, according to the options in the dump parameter. IBMBKDO uses either the declared PLITABS or loads the system default PLITABS module, IBMBSTAB, to determine the pagesize for PLIDUMP output. Provided a pagesize of two or more is specified, the pagesize in PLITABS will be used.

If the dump file can be successfully opened, IBMBKDO tests the attributes of the file. If it appears from the attributes that the dump is being transmitted directly to a printer or terminal, the transmitter IBMBKDT is loaded. If it appears that it is being transmitted to a direct-access device or tape unit, the transmitter IBMBKDB is loaded.

If IBMBKDT is loaded, two buffers are acquired. The address of one of these buffers is placed in the DUB. During the execution of the dump, the dump data is generated in the buffer which is addressed by the DUB. When the first buffer is full, a call is made to the transmitter module to transmit the buffer to the dump file. A test is then made to see whether the second buffer has completed the previous I/O operation. When the previous I/O operation (if any) is complete the address of the second buffer is placed in the DUB and the operation continues. If IBMBKDB is loaded, only one buffer is used.

When the dump is finished, the dump file remains open and the transmitter is retained. This speeds execution of further dumps. The storage is freed and the dump file closed by IBMBPIT when the program is terminated. The dump file is not placed on the open file chain. IBMBPIT tests the DUB to see if the file is open.

MISCELLANEOUS ERROR MODULES

A number of further library modules are used in certain exceptional error situations. These fall into two groups.

1. ABEND analyzers

   IBMBPES   Determine action to be taken.

   IBMBPEV   Put out message if necessary, and dump if possible.

2. Exceptional error message modules

   IBMBPEP   Exceptional error message director

IBMBPEQ   No main procedure or more than 1024 files and
          controlled variables.

IBMBPER   No main storage available

IBMBPET   Interrupt in error handling routines or abnormal
          task termination

All these modules are transient library modules. They are fully
described in the relevant program logic manual.

ABEND Analyzers

The ABEND analyzer IBMBPES is entered during an ABEND because it
was nominated in the STAE macro instruction issued during
program initialization.

The ABEND is analyzed by checking the major blocks to see if
they have been overwritten. If the back-chain of DSAs has
become overwritten, the ABEND is allowed to continue under
supervisor control. If the DSA back-chain is correct but
critical control blocks appear to be overwritten, IBMBPEV is
called to put out a message and if possible to provide a
PLIDUMP. If no overwriting is detected, the error handler is
called with a code indicating the error condition.

The message put out by IBMBPEV where possible contains the
number of the PL/I statement being executed when a ABEND
occurred.

EXCEPTIONAL ERROR MESSAGE MODULES

The exceptional error message modules consist of a director and
three message modules. This arrangement has been adopted so
that the minimum space will be used. It is necessary to
conserve space as lack of space is one of the reasons for
calling the modules.

The director module IBMBPEP determines the nature of the
exceptional error and calls the necessary module to put out the
message.

The table below shows the circumstances in which IBMBPEP is
called and message modules then called by IBMBPEP.

Circumstance	Calling Module	IBMBPEP Calls
Insufficient main storage to set up in the program management area	IBMBPII	IBMBPER
No main procedure	Code in dummy PLIMAIN	IBMBPEQ
Too many files, controlled variables, and fetched procedures to be held in PRV	IBMBPII	IBMBPEQ
Interrupt in error handling routine	IBMBERR entry point C	IBMBPET
Abnormal termination of task	IBMTPIR	IBMBPET

Module IBMBPEQ puts out the message to SYSPRINT except in those
circumstances where SYSPRINT cannot be used. IBMBPET and
IBMBPER always put out their messages on the console as they are
called in circumstances where SYSPRINT is likely to fail or
where operator, rather than programmer action, is required.

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

The compiler generates the same executable code for both the COUNT and the FLOW option. Consequently, if either option is specified for compilation, either or both can be made available at execution time. If neither is required during execution but one or other was specified for compilation, the code to call IBMBEFL is still executed and IBMBEFL still forms part of the load module. When IBMBEFL is called in this situation, it returns control to compiled code without recording any information.

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 IBMBEFL. If the branches are taken, they are recorded. For COUNT they are recorded in a table known as the <u>statement frequency count table</u>. For FLOW, they are recorded in a table known as the <u>flow statement table</u>. The format of these tables are shown in detail in Appendix A, "Control Blocks" on page 326.

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 IBMBEFL 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 55 on page 143.

Special cases: There are a number of special cases that require additional action, either by the compiler, or by IBMBEFL, 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 IBMBEFL 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 IBMBEFL 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 IBMBEFL to set appropriate flags when a GOTO out of block occurs. A test for the flags is made when the call to IBMBEFL 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 IBMBEFL testing for such code. Details of the methods used are given later.

IMPLEMENTATION OF FLOW AND COUNT

Tables Used by FLOW and COUNT

To enable it to retain FLOW and COUNT information, IBMBEFL sets up tables in dynamic storage. Figure 56 on page 144 shows their contents. Details of their formats are shown in Appendix A, "Control Blocks" on page 326.

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.

Note: In certain conditions similar code will compile so that statement 2 is executed three times. See section on DO-loops in Chapter 2, "Compiler Output" on page 12.

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

There is another branch into statement 2 followed by a branch out to statement 5 because I>2. The final table is therefore:

1	2	3	4	5
1	2	-1	0	-1

ANALYSIS OF THE STATEMENT FREQUENCY COUNT TABLE

A value known as the current, 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 55. How Branch Counts Are Used to Calculate the Number of Times Each Statement Is Executed

FLOW STATEMENT TABLE   **STATEMENT FREQUENCY COUNT TABLE**

One table for the Program

One Table for each
External Procedure

HEADER SECTION	Contains: . Pointers to next entry in each section of the table. . Other housekeeping data
STATEMENT NUMBER SECTION	Contains: . Statement numbers of branches plus flags indicating the type of entry.
PROCEDURE NAMES SECTION	Contains: Names of procedures and types of ON-units that have been branched to.

HEADER SECTION	Contains: . Pointer to any further tables. . Address of static control section of associated external procedure. Other housekeeping data.
BRANCH COUNT SECTION	Contains: . A field for each statement in the program containing a count which is incremented when the statement is branched to and decremented when the statement is branched from.

Figure 56.   The Contents of the Flow Statement Table and the Statement Frequency Count Table.

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.

The length of statement frequency count tables depends on
whether the GOSTMT or GONUMBER option is in effect. For GOSTMT
one fullword is used for each statement in the procedure. For

GONUMBER, two fullwords are used. This is because for GONUMBER
it is necessary to retain the statement number as well as the
count value. (For GOSTMT, the numbers will start at one and be
incremented by one and no record need therefore be kept.) If
neither GOSTMT nor GONUMBER is in effect, no attempt is made to
count the statements executed in the procedure, and a statement
frequency count table is not set up.

Executable Code for FLOW and COUNT

As described in the introduction, there are four stages in the
implementation of the FLOW and COUNT options. These are:

1. Action during compilation. The code to call the
    interpretive library routine IBMBEFL is placed at every
    predictable branch-in and branch-out point.

2. Action during program initialization. The necessary
    housekeeping fields are set up. This is done by the program
    initialization module IBMBPII and the flow module IBMBEFL
    called at entry point IBMBEFLA.

3. Action during execution. The branch-in and branch-out
    information is collected by entry point IBMBEFLB. Entry
    IBMBEFLC is also called to handle certain special cases.
    The call is made when the GOTO out-of-block code is
    executed.

4. Action during output. The necessary information is written
    out. This is done by IBMBESN for the FLOW option and
    IBMBEFC for the COUNT option.

These four stages are described in detail in the following
sections.

Action during Compilation

During compilation, the compiler examines the program and
generates suitable code at each predictable branch-in and
branch-out point. Predictable branch-in points are:

- Entry names

- Labeled statements

- THEN and ELSE clauses of IF statements.

- Entries to ON-units

- Returns from CALL statements or function references.

- The statement following the END statement of an internal
  procedure.

Predictable branch-out points are:

- GOTO statements

- Function references

- CALL statements

- IF statements

- RETURN statements

- END statements

- The statement before the PROCEDURE statement of an internal
  procedure.

The code for branch-out points is so placed that the call to
IBMBEFL will not be made unless the branch is taken.

Statements preceding and following internal procedures are
treated as branch-out and branch-in points because the statement
numbers of the statements executed are not sequential although
the actual flow of control is sequential. If this were not
done, the method used for counting statements would not work
because the statements in the internal procedure would be given
the count values of the preceding statements.

The code placed at the branch-in and branch-out points takes the
following form:

```
L        15,   84(0,12)        Pick up address of
                               IBMBEFL from TCA.

BALR          14,  15          Branch to IBMBEFL.

DC       X'8004'               Constant containing
                               a two-bit flag
                               remainder for
                               statement number.
```

Register 14 is set to the constant containing the statement
number and flags by the BALR instruction. IBMBEFL can therefore
pick up the statement number by examining the constant.

The constant is a halfword if the STMT option was used and a
fullword if the NUMBER option was used. In both cases, the
first two bits are used as flags and the remainder are used for
the statement number.

The flags indicate:

•   Branch-in

•   Branch-out

•   Branch-in to a new procedure or ON-unit.

•   Return to point of interrupt from end of ON-unit.

For a branch-in to a CALL statement or a function reference,
which takes place when the return is made, BAL 14, 0(15) is
generated instead of BALR 14, 15. This situation requires to be
recognized because the branch-in and branch-out both occur from
the same statement. If it were not treated as a special case,
the count of the next statement would be decremented by one when
the branch-out was made and the count for the CALL statement
would be incremented by one on return. Thus the CALL statement
would apparently have been executed twice. The increment is
therefore added to the statement after the CALL statement,
thereby giving the correct values.

In addition to the calls to IBMBEFL, the compiler also generates
control sections that will result in IBMBEFL being link-edited
and subsequently called during program initialization to set up
the necessary housekeeping machinery to handle COUNT or FLOW.

For the FLOW option, the compiler generates a control section
called PLIFLOW that can be used during program initialization to
call IBMBEFL. This control section takes the following form:

```
        USING   *,15
        L       15,VCON
        BALR    1,15
        DC      H'n
        DC      H'm
VCON    DC      V(IBMBEFLA)
```

For the COUNT option, the compiler generates a control section
called PLICOUNT that can be used to call IBMBEFL to initialize

the COUNT option. It is the same s PLIFLOW except that the
halfwords 'n' and 'm' are replaced by a fullword X'80000000'.

The calls to IBMBEFL are generated if either FLOW or COUNT is
defined at compile-time. The control sections are generated if
the corresponding option is specified at compile time.

Action during Program Initialization

During program initialization, the program initialization module
IBMBPII determines if either FLOW or COUNT or both are required.
If the user specified either FLOW or COUNT during compilation,
the requested option will be in effect during execution unless
specifically overridden by the NOFLOW or NOCOUNT execution time
option. If he specified either option for compilation he can
also specify the other for execution.

To determine which options are to be used, IBMBPII inspects the
execution time options and checks for the presence of PLIFLOW or
PLICOUNT which will indicate that the corresponding option was
requested at compile-time.

If one or both of the options are requested for execution but
neither was requested for compilation, IBMBPII generates a
message to say that the option will not be available.

If an option is specified for compilation and not overridden for
execution time, the corresponding control section will be
available and IBMBPII passes control to entry point IBMBEFLA
through the code in the control section. If the control section
corresponding to the required option does not exist, IBMBPII
calls IBMBEFL directly, passing it a value in register 0. This
value is 4 if FLOW is required and 8 if COUNT is required.

If one or both of the options are requested during compilation
but neither are required during execution, IBMBPII sets FLOW
values of (0,0) and calls entry IBMBEFLB to initialize the FLOW
option. In this situation, IBMBEFLA sets the address of the
flow statement table and the addresses of the statement
frequency count tables to zero.

To initialize FLOW, IBMBEFLA sets up the flow statement table
and initializes it with a dummy statement number entry and a
dummy procedure name entry. The address of the flow statement
table is placed in the TCA. If FLOW is not required, or if
FLOW(0,0) has been specified, the address is set to zero.

To initialize COUNT, two addresses in the TIA are initialized.
The first, which will contain the address of the first of the
chain of statement frequency count tables, is set to zero. The
second which will contain the address of the current statement
frequency count table is set to point to the first. If COUNT is
not required, both fields are set to zero.

For both FLOW and COUNT, the address of IBMBEFLB is placed in
the TCA; the GOTO code, which is in the TCA, is altered so that
it calls IBMBEFLC. (This is necessary so that changes of block
caused by GOTO statements can be intercepted and flagged.)

Action during Execution

During execution, calls from compiled code at branch-in and
branch-out points are made to IBMBEFLB whose address has been
placed in the TCA. The action then taken depends on which
options are in effect, the type of the previous entry, and the
type of the present entry.

Calls are also made to IBMBEFLC when the GOTO code in the TCA is
executed.

IBMBEFL When Called at Branch-In and Branch-Out Points

When IBMBEFL is called at branch-in or branch-out points, the call goes to entry point B whose address has been placed in the TCA during program initialization. IBMBEFL first checks to see which, if either, of the options is required by testing the fields used to address the flow statement table and the current statement frequency count table. If either of these is set to zero, the corresponding option is not in effect. If both are set to zero, control is returned to compiled code. If one or the other of the options is in effect, there are four possible cases that require different action:

1. A branch-in following a branch-out or vice versa

2. A branch-in following another branch-in

3. A branch-in to a new block

4. Return from an ON-unit to the point of interrupt

These cases are dealt with individually in the sections that follow.

CASE 1. BRANCH-IN FOLLOWING A BRANCH-OUT OR VICE VERSA: This situation indicates nonsequential flow of control, and must therefore be recorded in the FLOW and COUNT tables. For FLOW, the new statement number together with flags indicating a branch-in, a branch-out, or a branch-in to a procedure or ON-unit, are entered in the position indicated by the pointer at the head of the flow statement table. The pointer is then updated to point to the next available space. If the next space would be outside the table, the pointer is reset to the head of the statement number section of the table.

For COUNT, the count value in the field for the appropriate statement number is altered. For a branch-in, the count of the statement branched to is incremented by one. For a branch-out, the count of the statement after the statement branched from is decremented by one.

If IBMBEFL is being called for a branch-in, it is possible that it was caused by a GOTO-out-of-block and a new procedure or ON-unit name may need to be recorded. In this situation, IBMBEFLC will have been called during the execution of the GOTO-out-of-block code and will have set a flag in the flow statement table. The flag is therefore tested and, if it is found on, the entry is treated as an entry to a new block. See "Case 3. A Branch-In to a New Block" on page 149.

A further possibility is that the branch-in will be a returned to a CALL statement or a function reference. These are distinguishable because the call to IBMBEFL is made by a BAL instruction rather than a BALR instruction. If the COUNT option is in effect, this must be tested for, and the count value of the _next_ statement rather than the current statement be incremented. This is necessary because the branch-out and the branch-in for CALL statements and function references are both made at the same statement, (see the description under "Action during Compilation" on page 145).

CASE 2. A BRANCH-IN FOLLOWED BY ANOTHER BRANCH-IN: No action need be taken as such a situation can only be caused by sequential flow. For example consider the statements:

```
LAB1:    X=Y;
LAB2:    Z=X;
```

Both LAB1 and LAB2 are potential branch-in points, but, if a call to IBMBEFL is made for LAB2 immediately after a call has been made for LAB1, it is plain that the flow of control has been sequential. Consequently when a branch-in follows another branch-in, IBMBEFL returns control to compiled code without taking any action.

This situation does not arise with branch-out points, because the code to call IBMBEFL is only executed if the branch is taken.

CASE 3. A BRANCH-IN TO A NEW BLOCK: This case requires that block information be entered for the FLOW option, and that, for the COUNT option, a check be made to see whether a new external procedure has been entered. If it has, a different statement frequency count table will have to be used because there is one for each external procedure.

Special action will be required if the block entered is an ON-unit. This is because the branch-out will have been made at the point of interrupt and this will not have been automatically recorded by a call to IBMBEFL. When a new block is entered a test is therefore made on the DSA flags of the block to establish whether it is an ON-unit. The action taken if it is an ON-unit is described under "Branch-In to an ON-Unit" on page 150.

After any action required to handle entry into an ON-unit, the following will take place.

For FLOW, the name of the block must be discovered and placed in the next available space in the names section of the flow statement table. Also, the statement number entry must be flagged to show that it marks a change of block. The procedure name is found by following the DSA chain back until a procedure DSA is found and accessing the name, which is held at a standard offset from the entry point of the procedure. When the procedure name has been found, the statement number and flags, and the procedure name, are placed in the appropriate sections of the flow statement table and the pointers altered to point to the next available fields.

For COUNT, a check must be made to discover whether a new statement frequency count table is required. This is done by comparing the address in the register 3 save area of the DSA of the procedure that called IBMBEFL with that at offset X'4' in the current statement frequency count table. If they are the same no action is required, because the new block must have the same static control section as the previous block and consequently must be in the same external procedure. If the addresses are not the same, a search is made down the chain of statement frequency count tables for a matching table. If one is found, the address of the current table is set to point to the table that has been found, and the required entry is made in that table. If no matching table is found, a new table must be set up.

CREATING A NEW STATEMENT FREQUENCY COUNT TABLE: Before creating a new statement frequency count table, IBMBEFL checks to see if a statement number table exists for the new procedure. If it does not, counting will not take place. In this situation, the current statement frequency count table is flagged to indicate that counting is to be suspended until another procedure is entered, and control is returned to compiled code.

Provided a statement number table does exist, a new statement frequency count table will be required. IBMBEFL first obtains the required amount of non-LIFO storage for the table. One fullword is required for every statement in the external procedure if it was compiled with the GOSTMT option, and two fullwords are required for every statement if it was compiled with the GONUMBER option. The count fields are set to zero, and, for procedures compiled with the GONUMBER option the numbers are inserted in the tables. The new table is then linked with its matching external procedure by placing the address of the static control section for the procedure in the new table.

BRANCH-IN TO AN ON-UNIT: If the code that called IBMBEFL is
found to be in an ON-unit, special action is required. The
statement number for the point of interrupt must be discovered
and appropriate entries made in the flow and count tables,
before the data for the entry to the ON-unit can be recorded.
This is because there will have been no call to IBMBEFL at the
point of interrupt to register a branch-out. The statement
number of the interrupt is found by IBMBEFL in the same way as
that used by the error message modules, described earlier in
this chapter. When the number has been found, it is
incorporated in the flow and count tables as if it were a normal
branch-out. The branch-in entry is then handled as if it were a
normal entry to a new block. It is possible for the FLOW option
to be in effect without there being a statement number table
available. In this situation, a statement number of zero is
entered in the flow statement table for the branch-out at the
point of interrupt.

A problem also exists for COUNT if an interrupt results in the
termination of a program. In this situation, the interrupt
point must be marked as a branch-out, otherwise, statements
after the interrupt would have an incorrect count value. This
situation is checked for when the FINISH condition is raised.
During the handling of the FINISH condition, the GOTO code is
executed and IBMBEFLC is called. A check is then made to see if
FINISH was raised because of an interrupt. If it was, the point
of interrupt is discovered and entered as a branch-out point in
the appropriate statement frequency count table.

CASE 4. RETURN FROM ON-UNIT TO POINT OF INTERRUPT: When return
is made from the end of an ON-unit to the statement that caused
the interrupt, there will be no automatic call (resulting from
code inserted during compilation) to IBMBEFL. The necessary
information for the flow and statement frequency count tables is
therefore entered when IBMBEFL is called at the end of the
ON-unit. The statement numbers passed for such calls are
specially flagged so that IBMBEFL discovers the point of
interrupt and takes the necessary action to update the flow
statement table and statement frequency count tables.

Action on Output

Interpreting the Flow Statement Table

Information from the flow statement table is interpreted by the
message module IBMBESN 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 IBMBEFC is called at program termination to print count
information. Output is tabular and printed three 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 heading: "FROM TO COUNT." The current
count is initialized to zero and the first nonzero 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
nonzero entry is added to the current count. The entries for
the following statements are scanned until one with a nonzero
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 nonzero 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.

ERROR HANDLING UNDER CICS

Error handling is managed by the module IBMFERRA which is
included in DFHSAP. The STAE/NOSTAE execution time option is
used to determine whether PL/I will handle program checks (or
ASRA ABENDs as they are known in CICS).

If STAE is specified, PL/I attempts to handle program checks.
It does this by issuing an EXEC CICS HANDLE ABEND command during
program initialization. Two transient modules are used to
handle messages; IBMFESMA and IBMFESNA. Full details of the
modules are given in the Resident and Transient Library PLMs.

PLIDUMP ON CICS

PLIDUMP is implemented by a set of modules based on the system
used in OS.

The modules are:

Resident

IBMBKDMA   Resident dump bootstrap routine in DFHPL1OI

COMPILATION

→ Ensure program tests for all enabled non-system-detected interrupts and calls error handler when they occur → Compile ON-units as separate program blocks

INITIALIZATION

Issue SPIE/ESPIE and STAE/ESTAE macros to pass system detected interrupts to correct error handler →

PROLOG CODE

→ Set up flags to indicate to error handler which conditions are enabled → Mainline code →

PROCEDURE CODE

Execute ON statement by setting flag indicating ON-unit is established → Mainline code →

PROCEDURE CODE

→ Instruction capable of causing condition not detectable by system → Test whether condition occurred — Mainline code → NO / YES

(A) Mainline code

System detected interrupt → Set up PSW branch to error handler because of SPIE/ESPIE or STAE/ESTAE

(A) Mainline code

OBJECT PROGRAM

Prepare data including condition type and built in function values

Call error handling module IBMBERR

Determine PL/I condition from PSW

IBMBERR
Error handling module
ENTRY POINT A

ENTRY POINT B

IBMBERR
Error handling module

Is it the ERROR cond — YES → Call message module to put out message

NO

Return to point of interrupt ← YES — Is condition disabled

NO

If no GOTO out of ON-unit take action for normal return. (See below) ← Execute action specified in ON-unit ← YES — ON-unit established; correct AMODE ?

NO → Take standard system action (See below)

Possibly put out message, then take one of actions shown below depending on condition that occurred

| Return to point of interrupt | Call IBMBPIR to terminate program or task | Raise further condition |

(A)

Figure 57. Outline of Error Handling

Transient

IBMFKMRA	Dump control module
IBMFKPTA	Process dump options
IBMFKTCA	Check for back-chain errors
IBMFKTRA	Handle T (Trace) option
IBMFKTBA	Handle B (Block) option
IBMFKCSA	Handle K (CICS) option

All routines use a transmitter in IBMFKMRA to print the output on the queue with the ddname of CPLD. Before the printing starts, an ENQ macro is issued so that two or more dumps will not be printed at once.

Each of the modules in the chain deletes its predecessor unless the predecessor is the control routine IBMFKMRA. The arrangement of modules is shown in Figure 58.

The back-chain check module IBMFKTCA is used to check that the back-chain has not been overwritten. It it has, an indicator is set and the other modules avoid the errors that this might cause.

The dump output is headed by the terminal identifier, transaction identifier, and date and time of execution.
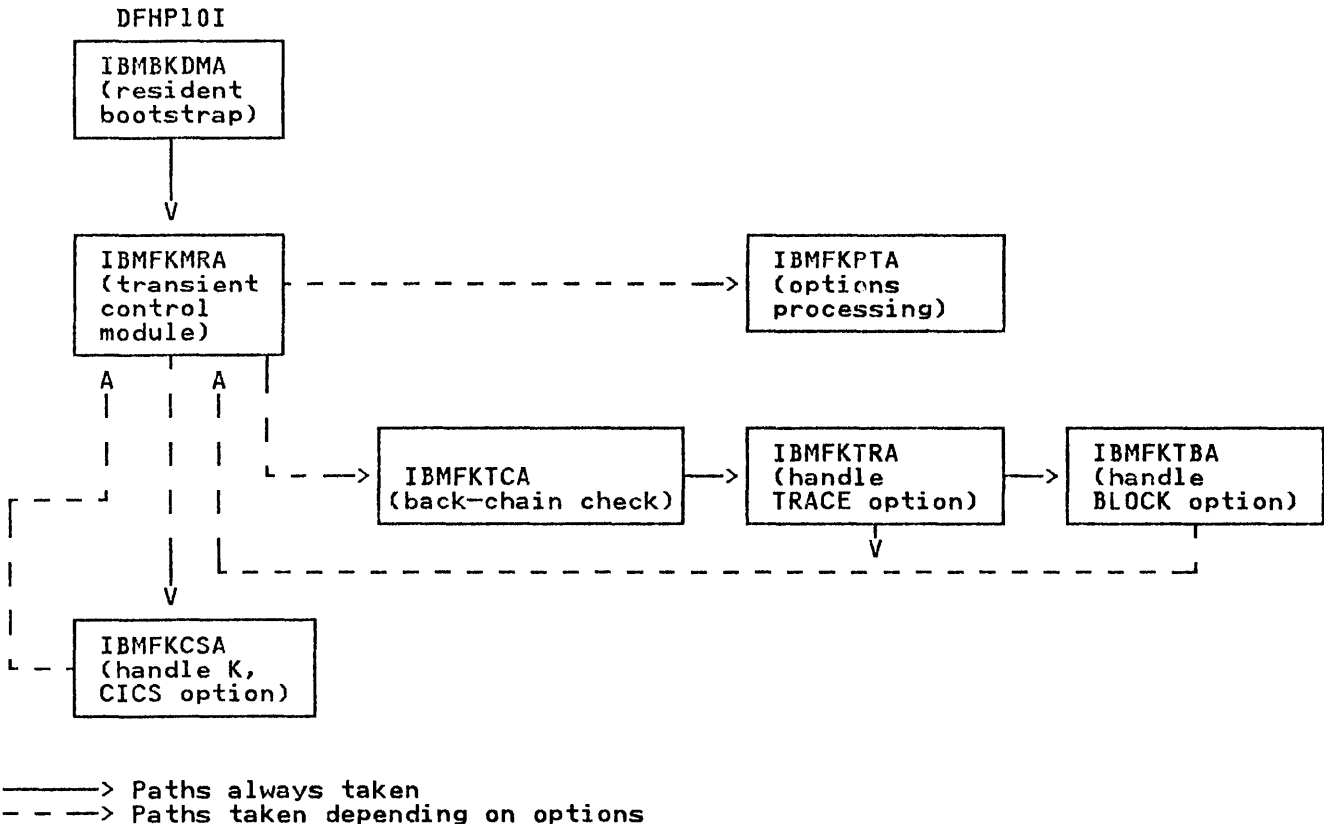
Figure 58. The Arrangement of PLIDUMP Modules for CICS

CHAPTER 8. RECORD-ORIENTED INPUT/OUTPUT

INTRODUCTION

This chapter considers the implementation of the following statements:

- File declarations

- Open and close statements

- READ, WRITE, DELETE, LOCATE, UNLOCK, and REWRITE statements referred to generically as transmission statements

Together, these statements make up record I/O.

The OS PL/I Optimizing Compiler uses the data management routines of MVS to implement record I/O. These routines offer facilities similar but not identical to those of the PL/I language. The data management routines require that:

1. A data control block (DCB) 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 normally issued to store or obtain a new record.

The data management 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 data management 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 data management routines in the appropriate manner. The method is summarized below, and shown diagrammatically in Figure 59 on page 155. Figure 73 on page 183 shows the overall scheme in greater detail.

SUMMARY OF RECORD I/O IMPLEMENTATION

File Declarations

For a file declaration, the compiler generates two control blocks: the declare control block (DCLCB) and the environment control block (ENVB). Together, these two control blocks contain a complete record of the file declaration.

```
COMPILER                    ┌─────────────────────────────┐
                            │ Set up control blocks        │
                            │ from file declaration        │
                            │ and I/O statements           │
                            └─────────────────────────────┘
─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─

COMPILER GENERATED CODE     ┌─────────────────────────────┐
                            │ Call PL/I library or data    │
                            │ management routines          │
                            │ passing control blocks       │
                            └─────────────────────────────┘

              OPEN & CLOSE STATEMENTS        TRANSMISSION STATEMENT

                            In-line I/O              Library Call I/O
─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─
                                                          PL/I LIBRARIES

                V                                    V
┌──────────────────────────────────────┐   ┌──────────────────────────┐
│ OPEN/CLOSE BOOTSTRAP                   │   │ TRANSMITTER INTERFACE     │
│ ROUTINE                                │   │ ROUTINE                   │
│                                        │   │                           │
│ (Resident library)                     │   │ (Resident library)        │
└──────────────────────────────────────┘   └──────────────────────────┘
       V                    V                    V              V
┌──────────────────┐  ┌──────────────────┐  ┌──────────────────────────┐
│ OPEN ROUTINES     │  │ CLOSE ROUTINE     │  │ PL/I TRANSMITTER          │
│                   │  │                   │  │                           │
│ (Transient library)│ │ (Transient library)│ │ (Transient library)       │
└──────────────────┘  └──────────────────┘  └──────────────────────────┘
─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─
                                                  DATA MANAGEMENT
                                                  ROUTINES
                                                  OPERATING SYSTEMS
       V                    V                    V
┌──────────────────┐  ┌──────────────────┐  ┌──────────────────────────┐
│      OPEN         │  │     CLOSE         │  │ DATA MANAGEMENT           │
│    ROUTINE        │  │    ROUTINE        │  │ TRANSMITTER ROUTINE       │
└──────────────────┘  └──────────────────┘  └──────────────────────────┘
```

Figure 59. The Principles Used in Record I/O Implementation

OPEN Statements

OPEN statements are compiled as a call to a resident-library bootstrap routine, IBMBOCL, which has passed to it an open control block (OCB) containing the attributes and environment options that have been used in the OPEN statement.

The bootstrap routine loads and calls a number of transient routines that build a definitive control block, known as the file control block (FCB), from information in the DCLCB, ENVB, and OCB. The file is associated with the data set, and the appropriate PL/I transmitter module is loaded.

The FCB is used during the execution of transmission statements to access all file information. It is addressed via the DCLCB and the pseudo-register vector.

Transmission Statements

For the majority of file and statement types, details of statement type, of record, key, and event variables are set up in control blocks during compilation; during execution, these control blocks are passed to a resident-library interface routine, IBMBRIO. IBMBRIO then calls a PL/I transient-library transmitter module, which issues the appropriate data management macro instruction, and checks for errors, before returning control to compiled code. This method is known as library-call I/O.

If the TOTAL option is used, the majority of transmission statements on buffered consecutive files are compiled as short calls to the data management routines. This method is known as in-line I/O. When using in-line I/O, subroutines of the PL/I transmitters are use to branch directly to the data management routines. When running in an MVS/XA environment, the subroutines set the correct addressing mode (AMODE) for data management. These transmitters are also used for error situations and end-of-file conditions.

The TOTAL option is a method used to inform the compiler that no additional information will be supplied about the file via the DD statement. (That is, that the TOTAL information about the file has been declared.) This allows the compiler to determine whether or not inline I/O statements can be used. The conditions when they are used are described in Figure 74 on page 184.

CLOSE Statements

CLOSE statements are implemented by a call to the open/close bootstrap routine IBMBOCL, which loads and calls the transient close routine IBMBOCA. This routine disassociates the file from the data set, and handles the necessary housekeeping.

Implicit Open

Implicit opening is handled by manipulation of addresses in the file control block (FCB). Any attempt to access the file when it is not open results in control being passed to the open routines in the PL/I libraries. The FCB is mapped in "File Control Block (FCB)" on page 360.

Implicit Close

Implicit closing is handled by the program termination routine checking for open files, and if it finds any, calling the PL/I library routine to close them.

As can be seen from the summary above, a large number of library subroutines and control blocks are used in the implementation of record I/O. These are summarized in two figures: Figure 60 for library subroutines and Figure 62 on page 160 for control blocks. More detailed descriptions for each statement type are given below.

RESIDENT LIBRARY

IBMBOCL   Open/Close bootstrap routine
IBMBRIO   Record I/O interface routine

TRANSIENT LIBRARY

Open Modules

IBMBOPA   Open error handler
IBMBOPB   Open routine Phase I
IBMBOPC   Open routine Phase II
IBMBOPD   Open routine Phase III
IBMBOPE   Open routine Phase II (VSAM)
IBMBOPZ   Direct output file formatter

Close Module

IBMBOCA   Close module

Transmitter Modules

IBMBRAA   Regional sequential output
IBMBRAB   Regional sequential output
IBMBRAC   Regional sequential output
IBMBRAD   Regional sequential output
IBMBRAE   Regional sequential output
IBMBRAF   Regional sequential output
IBMBRAG   Regional sequential output
IBMBRAH   Regional sequential output
IBMBRAI   Regional sequential output
IBMBRBA   Regional sequential input/update
IBMBRBB   Regional sequential input/update
IBMBRBC   Regional sequential input/update
IBMBRBD   Regional sequential input/update
IBMBRBE   Regional sequential input/update
IBMBRBF   Regional sequential input/update
IBMBRBG   Regional sequential input/update
IBMBRCA   Unbuffered consecutive
IBMBRCB   Unbuffered consecutive
IBMBRCC   Unbuffered consecutive
IBMBRCD   Unbuffered consecutive OMR
IBMBRCE   Unbuffered consecutive associated file
IBMBRDA   Regional direct non-exclusive
IBMBRDB   Regional direct non-exclusive
IBMBRDC   Regional direct non-exclusive
IBMBRDD   Regional direct non-exclusive
IBMBRJA   Indexed sequential input/update
IBMBRJB   Indexed sequential input/update
IBMBRKA   Indexed direct non-exclusive
IBMBRKB   Indexed direct non-exclusive
IBMBRKC   Indexed direct non-exclusive

Figure 60 (Part 1 of 2). Library Subroutines Used in Record I/O

Transmitter Modules (Continued)

IBMBRLA	Indexed sequential output
IBMBRLB	Indexed sequential output
IBMBRQA	Buffered consecutive (non-spanned)
IBMBRQB	Buffered consecutive (non-spanned)
IBMBRQC	Buffered consecutive (non-spanned)
IBMBRQD	Buffered consecutive (non-spanned)
IBMBRQE	Buffered consecutive input (spanned)
IBMBRQF	Buffered consecutive output (spanned)
IBMBRQG	Buffered consecutive update (spanned)
IBMBRQH	Buffered consecutive OMR
IBMBRQI	Buffered consecutive associated file
IBMBRTP	Teleprocessing file input
IBMBRVA	VSAM ESDS transmitter
IBMBRVG	VSAM KSDS sequential output
IBMBRVM	VSAM KSDS other operations and path
IBMBRVI	RRDS
IBMBRXA	Exclusive regional direct update update/input
IBMBRXB	Exclusive regional direct update update/input
IBMBRXC	Exclusive regional direct update update/input
IBMBRXD	Exclusive regional direct update update/input
IBMBRYA	Exclusive indexed direct update update/input
IBMBRYB	Exclusive indexed direct update update/input
IBMBRYC	Exclusive indexed direct update update/input
IBMBRYD	Exclusive indexed direct update update/input
IBMBSOF	Stream output file
IBMBSOU	Stream output file
IBMBSOV	Stream output file
IBMBSTF	Stream output print file
IBMBSTI	Stream input file
IBMBSTU	Stream output print file
IBMBSTV	Stream output print file
IBMCSTI	Stream input file
IBMCSTP	Stream output file

Record I/O Error Modules

IBMBREA	Record I/O error module
IBMBREB	Record I/O error module
IBMBREC	Record I/O error module
IBMBREE	Record I/O error module
IBMBREF	Record endfile module

Figure 60 (Part 2 of 2). Library Subroutines Used in Record I/O

The access method used for different PL/I file types is shown in Figure 61.

File Type	Access Method
Buffered consecutive	QSAM/VSAM
Unbuffered consecutive	BSAM/VSAM
Regional sequential (not spanned records)	BSAM
Regional sequential (spanned records only)	BDAM
Regional direct	BDAM
Indexed sequential	QISAM/VSAM
Indexed direct	BISAM/VSAM
TP buffered input/update	TCAM
VSAM	VSAM

Figure 61. Access Methods and File Types

Consecutive or indexed files can be used to access VSAM data sets; the PL/I open routines will determine the data type. For details see section on OPEN statement.

CONTROL BLOCKS GENERATED FROM FILE DECLARATION	CONTROL BLOCK GENERATED FROM OPEN STATEMENT

DCLCB

```
Function: Holds all file attributes
used in file declaration

Location: Separate control section
for external files, static internal
for internal files

When generated: During compilation

Contents:
   Record of file attributes
   at declaration
   File name
   Address of ENVB
   Offset of FCB pointer in PRV
```

Open control block (OCB)

```
Function: To contain file attributes
given in OPEN statement

Location: In static storage

When generated: During compilation

Contents: The attributes when
specified on the OPEN statement
```

Environment control block (ENVB)

```
Function: Holds information on
environment options

Location: In static storage

When generated: During compilation

Contents: Addresses of
   blocksize
   record length
   number of buffers
   KEYLOC value
   key length
   indexarea size
   addbuf
```

Figure 62 (Part 1 of 2). The Fields Used in Implementing Record I/O

CONTROL BLOCKS GENERATED FROM INPUT/OUTPUT STATEMENTS	CONTROL BLOCK GENERATED DURING EXECUTION OF OPEN STATEMENT
Key descriptor (KD)	File control block (FCB)

Key descriptor (KD)

```
Function: To describe the key
variable

Location: Depends on storage class
of key variable

When generated: Depends on
storage class of key variable
Contents: Length and address of
key variable
```

Record descriptor (RD)

```
Function: To describe the record
variable
Location: Depends on storage
class of record variable
When generated: Depends on
storage class of record variable
Contents: Length and address of
record variable
```

Request control block (RCB)

```
Function: Holds a definition of the
statement for execution-time checking
Location: In static storage
When generated: During compilation,
for library data management calls only
Contents: Flags defining statement
        Code for TM instruction,
        or a branch instruction
        (if checking was done
        during execution)
```

File control block (FCB)

```
Function: Acts as a central source
  of information about the file

Location: In static storage

When Generated: During open

Contents include:
  Flags indicating valid statements
  Transmitter name
  Transmitter address
  Error module address
 ┌─DCB/ACB address
 │ Filename address
 │ Buffer address flags and
 V workspace for the transmitters
 DCB
    Data Management control block/
    Access-Method Control Block
```

Figure 62 (Part 2 of 2). The Fields Used in Implementing Record I/O

FILE DECLARATION STATEMENTS

For each file declaration, a declare control block (DCLCB) and, optionally, an environment control block (ENVB) are set up. Both are held in static internal storage for internal files, or in a separate control section for external files.

The DCLCB is a control block that contains the filename together with a record of the attributes obtainable from the file declaration, both those given explicitly and those deducible by default. This information is retained until the file is opened, when, unless the TOTAL option has been used in the file declaration, the information is merged with any attributes in the OPEN statement.

The ENVB contains the addresses of all environment options. The format of the ENVB is shown in "Environment Block (ENVB)" on page 351.

From information in the DCLCB and the ENVB, (and sometimes from the open control block (OCB) produced from the OPEN statement) a further control block, the file control block (FCB) is generated. During execution of an I/O statement, all information about the file is derived from the FCB.

Execution

No executable code is produced from the file declaration. Figure 63 shows the code resulting from a file declaration.

```
DCL FI FILE UNBUFFERED RECORD INPUT          ENVIRONMENT (RECSIZE (80));
   _____|_____          _____|____
                           |                                              |
                           V                                              |
                         DCLCB                                            |
                           |                                              |
           _____|_____                             |
          |                                 |                             |
          0000000002010200                                               |
          0106190000000018                                               V
          000000140002C6F1
          0000000000000000
          0000000200000040
          0000004400000040   |<--ENVB--<--------------------------------
          0000004000000040
          0000004000000040
```
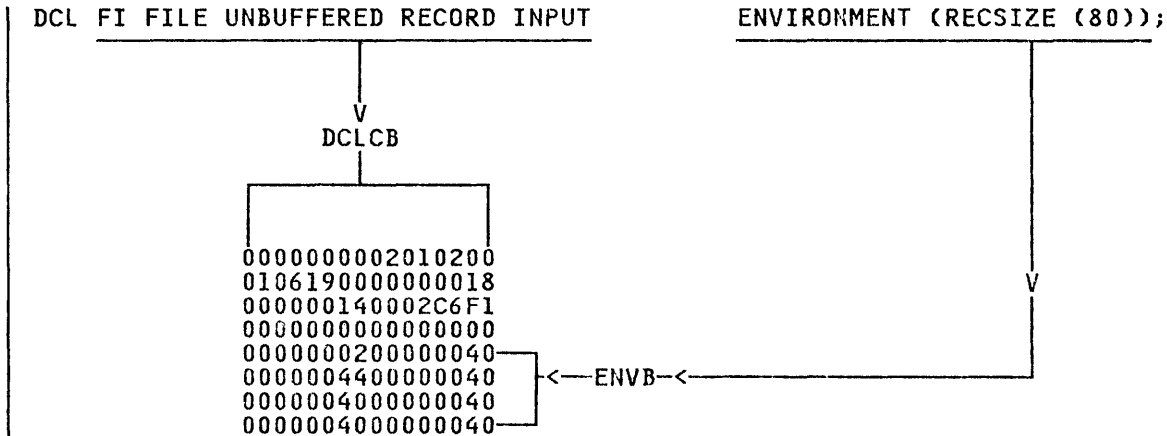
Figure 63.   Information in the File Declaration Is Held in the ENVB and the DCLCB Until the File Is Opened

OPEN STATEMENT

Compiler Output

For an OPEN statement, the compiler generates a call to the open/close bootstrap routine, IBMBOCL, and an open control block (OCB). The OCB holds any attributes that are declared in the OPEN statement.

More than one file may be passed to the open routines. The last file has its last parameter flagged with its first bit set to '1'.

Execution

For an explicit open, a call is made to the open/close bootstrap routine, IBMBOCL. For each file to be opened, the following information is passed to IBMBOCL:

    The address of the DCLCB
    The address of the OCB (or zero, if no OCB exists)
    The address of the TITLE (or zero, if none is specified)

IBMBOCL has four entry points:

    IBMBOCLA explicit open
    IBMBOCLB explicit open for library call I/O
    IBMBOCLC explicit close
    IBMBOCLD implicit close

When called by entry point A, IBMBOCL invokes the transient
library open routines to open the file. If the environment
option TOTAL has not been used in the file declaration, it will
be necessary to determine the attributes of the file by merging
the attributes in the file declaration with those used in the
OPEN statement. Attributes in the file declaration are held in
the ENVB and DCLCB. Attributes used in the OPEN statement are
held in the OCB. If the TOTAL option has been used, attributes
are taken from the declaration, and any contradictory attributes
in the OPEN statement result in the raising of the ERROR
condition.

The open modules build an FCB and DCB from the information in
the control blocks, initialize the pseudo-register vector to
point to the FCB, load the PL/I and data management
transmitters, and return to compiled code. File transient open
modules are used. Their functions are summarized below and are
described in detail in the licensed publication OS/360 PL/I
Transient Library: Program Logic.

Actions Carried Out by Transient Open Routines

The transient open routines perform the following major
functions when opening a file:

1. Build the file control block (FCB) and data control block
   (DCB), or, for VSAM the access method control block (ACB)
   for the file. The FCB is a PL/I control block used to
   access all file information. The DCB is a data management
   control block used to describe the data set. The ACB is the
   equivalent of the DCB for VSAM files.

2. Issue the data management OPEN macro instruction to
   associate the file with the data set.

3. Obtain and initialize buffers and any other blocks required
   for the file.

4. Determine which statement types are valid for the file, and
   store this information as a set of flags held in the FCB.

5. Select the appropriate PL/I transmitter, and load it for use
   during transmission statements.

6. Check for errors, and raise the UNDEFINEDFILE condition if
   any are found.

7. Place the address of the FCB in the correct pseudo-register
   vector offset.

The execution of an OPEN statement is summarized in Figure 64 on
page 164.

VSAM Data Sets

VSAM data sets, both KSDS and ESDS, are normally accessed by
PL/I using VSAM macro instructions, however, in certain
circumstances the data sets are accessed through the
compatibility interface. If the file is declared with ENV
(VSAM) the VSAM macro instructions will automatically be used.
Even if it is not so declared, the PL/I open modules will
normally detect that a VSAM data set is being accessed. To do
this they issue an RDJFCB macro instruction. However, this
action is not effective if the ALLOCATE command is being used
under TSO to provide DD information, because, in this case, the
RDJFCB macro instruction cannot determine that a VSAM data set
is being accessed. In this situation the compatibility
interface will be used. It is possible for the user to force
the use of the compatibility interface by specifying either
"RECFM" or "OPTCD=L" in the AMP parameter of the DD statement.

① DCLCB identifies file **②** Open control block (OCB) holds options in OPEN statement **③** Title held in static

OPEN FILE(F2) OUTPUT TITLE ('OUTFILE');

④ Executable instructions call to Open close bootstrap module passing parameter list **⑤** containing addresses etc
for **①** **②** and **③**

---
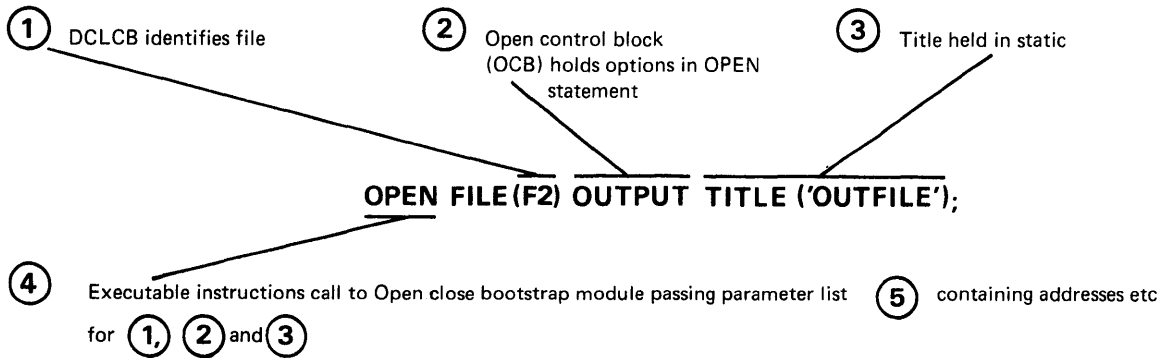
① DCLCB set up during file declaration see figure 8.5

② Open control block in static. See Appendix A for Format

```
000048   0020000000D00800 CONSTANT
         00000000
```

③ Title (held in static internal) is addressed via locator (also in static internal)

Title

```
0000A0   D6E4E3C6C9D3C5
```

Locator

```
000020   000000A000070000
```

④ Machine Instructions

```
000088 41 10 3 064   LA    1,100(0,3)        Point R1 at P-lists
00008C 58 F0 3 00C   L     15,A..IBMBOCLA  ⎫
000090 05 EF         BALR  14, 15           ⎬ Branch to open/close bootstrap
                                            ⎭
```

⑤ Parameter list

```
000064 00000044   A..CONSTANT        No. of files to be opened
000068 00000000   A..DCLCB
00006C 00000048   A..CONSTANT        A...OCB
000070 00000020   A..CONSTANT        A...LOCATOR for TITLE
000074 00000000   A..NULL ARGUMENT ⎫ Used for print files only
000078 80000000   A..NULL ARGUMENT ⎭
```

```
From          To
compiled      compiled                        EXECUTION
code          code
  ↓            ↑
┌──────────────┐   ┌──────────────┐  ┌──────────────┐  ┌──────────────┐  ┌──────────────┐
│ IBMBOCL      │   │ IBMBOPA      │  │ IBMBOPB      │  │ IBMBOPC      │  │ IBMBOPD      │
│──────────────│   │──────────────│  │──────────────│  │──────────────│  │──────────────│
│ Loads transient│ │              │  │              │  │              │  │              │
│ open modules.│   │ Open Phase I │  │ Open Phase II│  │ Open Phase III│ │ Open Phase IV│
│ Calls IBMBOPA│   │              │  │              │  │              │  │              │
└──────────────┘   └──────────────┘  └──────────────┘  └──────────────┘  └──────────────┘
                      ↕                  ↕
                   ┌──────────────┐  ┌──────────────┐
                   │ IBMBOPE      │  │ IBMBOPZ      │
                   │──────────────│  │──────────────│
                   │ Open Phase II│  │ Formatting   │
                   │ VSAM files   │  │ (direct output│
                   │              │  │ only)        │
                   └──────────────┘  └──────────────┘

RESIDENT LIBRARY          TRANSIENT LIBRARY
```
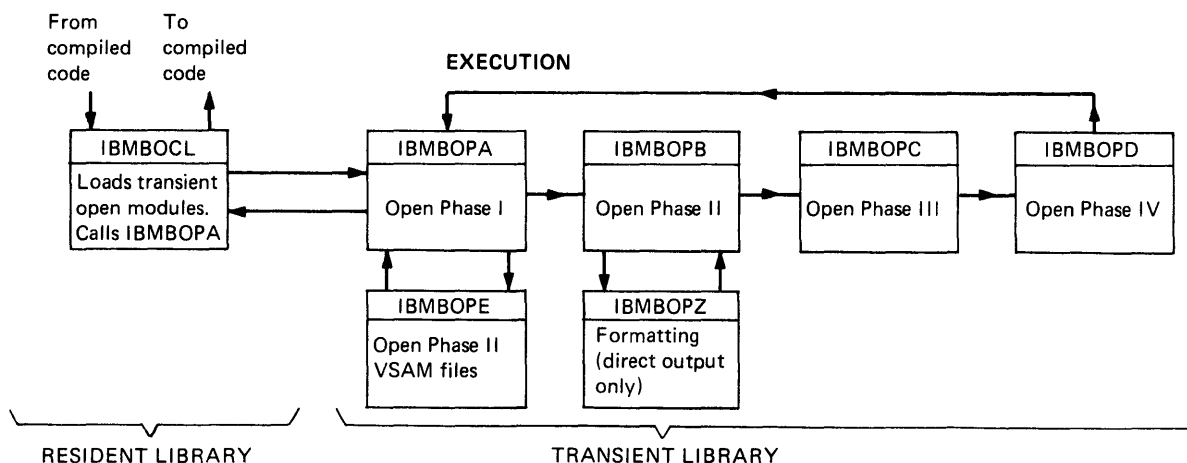
Figure 64.   OPEN Statement

The flow through the PL/I open modules is as follows. IBMBOPA
scans the list of files to be opened and sets a flag to indicate
that IBMBOPE is required for any files declared with ENV (VSAM).
If one or more files are found without ENV (VSAM), IBMBOPB is
called to open them. Then on return from IBMBOPB, IBMBOPE is
called to open any VSAM files. If IBMBOPB detects that any
consecutive or indexed files are being used to access VSAM data
sets, it will set the flag indicating that IBMBOPE is required
and ignore that file. When all the non-VSAM files have been
opened, IBMBOPD returns to IBMBOPA. IBMBOPA tests to see
whether there are any VSAM files to be opened, and, if there
are, calls IBMBOPE.

IBMBOPE opens the files starting with the first. Each file is
completely opened before starting to process the next. The open
process involves nine main steps, as follows:

1. Merge attributes from OPEN statement with file declaration
    and check for validity.

2. Get non-LIFO storage space for the FCB and ACB, and create
    the ACB using the GENCB macro instruction. The DDNAME is
    obtained from the filename or the TITLE option. The
    password is obtained from the PASSWORD environment option if
    specified.

3. Issue an OPEN macro instruction and test the return codes in
    the ACB.

4. Check the actual values of the RECSIZE, KEYLENGTH, and
    KEYLOC options against any values specified in the
    ENVIRONMENT option. Check that NCP/STRNO is not greater
    than one. If any errors or discrepancies are found, the ACB
    must be closed.

5. Set up the mask of invalid statements for use by IBMBRIO.

6. Get non-LIFO storage space for the IOCB and RPL, plus key
    space for a KSDS, and a dummy buffer for a buffered file.
    Create the RPL using a GENCB macro instruction.

    The OPTCD values are partially set as shown below. The
    transmitter merges the other options according to statement
    type. The OPTCD options set are:

        KEY/ADR   KSDS/RRDS/ESDS
        SEQ/DIR   SEQUENTIAL/DIRECT
        KSDS or PATH   INPUT/UPDATE/DIRECT
        UPD/NUP   UPDATE/INPUT or OUTPUT
        GEN/FKS   GENKEY/not GENKEY

    KEQ, MVE, and SYN are always specified.

7. Load the appropriate library transmitter as follows:

        ESDS   IBMBRVAA
        KSDS SEQUENTIAL OUTPUT
               IBMBRVGA
        KSDS SEQUENTIAL INPUT/UPDATE
        DIRECT/PATH
               IBMBRVHA
        KSDS DIRECT
               IBMBRVIA

8. Insert "E" as the seventh character of the error module
    name, so that IBMBREEA will be loaded if an error occurs.

9. Add the FCB address to the chain of open files and set the
    address of the FCB in the pseudo-register.

The FCB and File Addressing

During execution of record I/O statements, all information about the file is obtained from the FCB. However, as the FCB is not created until execution, the FCB cannot be addressed directly by compiled code. Instead, compiled code obtains from the DCLCB the offset within the PRV at which the FCB address is held. This offset is placed in the DCLCB by the linkage editor. The mechanism is illustrated in Figure 65.

The use of the pseudo-register vector allows separately compiled programs to refer to the same FCB for an external file, even though the address of the FCB cannot be known until execution. An explanation of the use of the pseudo-register vector is given in Chapter 2, "Compiler Output" on page 12, under the heading "The Pseudo-Register Vector."

```
                        TCA
    R12──>  ┌─────────────────────┐
            │                     │
            ├─────────────────────┤
            │  Address of PRV  ••••┐
            ├─────────────────────┤ •
            │                     │ •
            │                     │ •
            └─────────────────────┘ •
                                    •
                                    •
                                    •
                                    •
                                    •
                                    •
                                    •                      PRV
  DCLCB for file C                  •     V     ┌──────────────────────────────┐
  ┌─────────────────────────┐      •    ┌──┐ A │ Address of FCB for file A     │
  │  PRV offset = 8    <──┐  •••••>•    │  │ │ ├──────────────────────────────┤
  ├──────────────────────┼──┤      •    │  │ V │ Address of FCB for file B     │
  │                      │  │      •    └──┘   ├──────────────────────────────┤
  │                      │  │  ••••••>│ Address of FCB for file C     ┐
  │                      │  │         └──────────────────────────────┘ │
  │                      │  │                                          │
  │                      │  │              FCB for file C              │
  └──────────────────────┴──┘         ┌──────────────────────────────┐<┘
                                       │                              │
                                       │                              │
                                       └──────────────────────────────┘
```
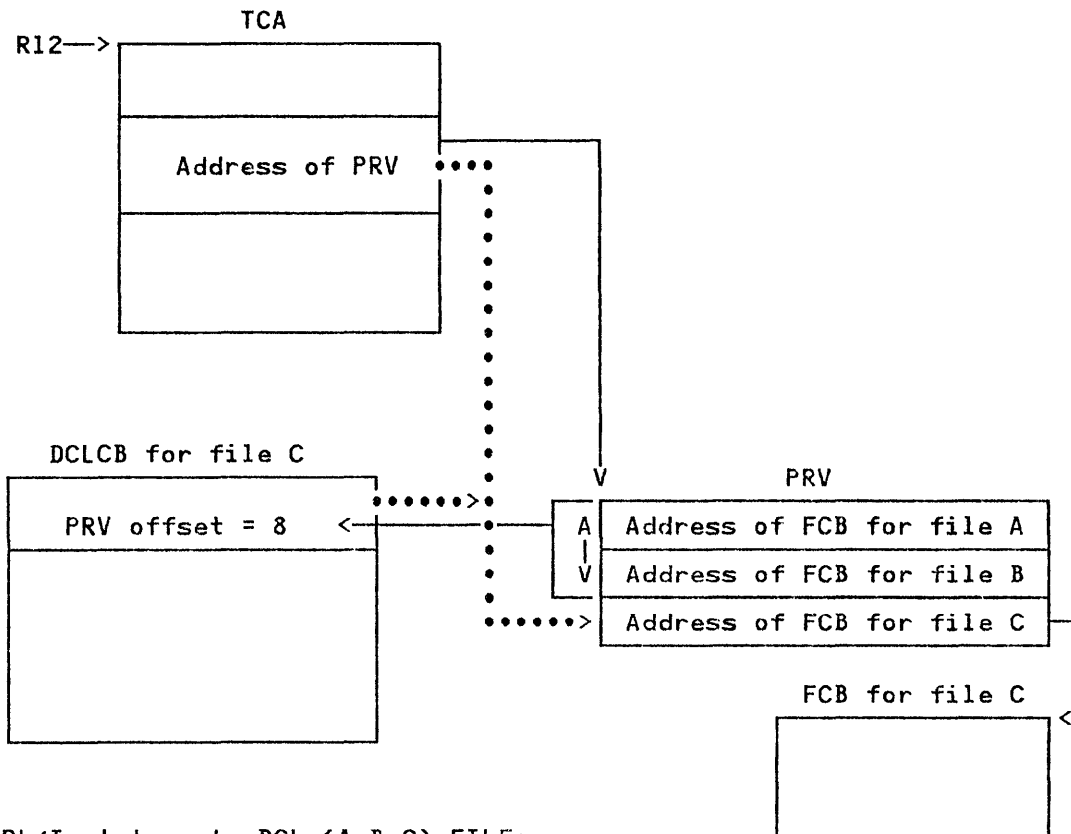
PL/I statement: DCL (A,B,C) FILE;

The address of the FCB for the file is obtained by adding the offset in the DCLCB to the PRV address which is held in the TCA

Figure 65. Addressing Files Via DCLCB and PRV

TRANSMISSION STATEMENTS (LIBRARY-CALL I/O)

Compiler Output

For transmission statements the compiler generates a call to the PL/I transmitter interface module, IBMBRIO. IBMBRIO has the following parameter list passed to it:

Address of DCLCB

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 DCLCB is generated from the file declaration, as described earlier in the chapter. The remainder of the control blocks in the parameter list are generated for the transmission statement.

The request control block (RCB) defines the statement type. It consists of two words. The first is a fullword of flags that define the statement type and option, indicating whether the statement is READ SET, READ INTO, WRITE FROM, etc. The second word is a test-under-mask (TM) instruction that is executed by IBMBRIO to check whether the statement is valid. The flags in the RCB are tested against flags in the FCB or dummy FCB. If the statement is invalid, a branch is made to an address held in either the FCB or the dummy FCB. If the file is not open, the dummy FCB will be accessed, and the branch will be made to the open/close bootstrap to open the file. If the file is open, a real FCB will be accessed, and the branch will be via a bootstrap to the error handler. The RCB is set up in static internal storage. The format is shown in "Request Control Block (RCB)" on page 391.

The record descriptor (RD) contains the address, length and type of the record variable. (The record variable is the variable to or from which the record will be transmitted.) A record descriptor is generated only if a record variable is used. The format is shown in "Record Descriptor (RD)" on page 390.

The key descriptor (KD) contains the address and length of the key variable. (The key variable is the variable to or from which the key will be transmitted.) It is generated only if a key variable is used. The format is shown in "Key Descriptor (KD)" on page 378.

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, and will be completed by the inclusion of the address during execution. The completed descriptor may be moved into temporary storage. In certain conditions, no skeleton is produced; instead, the complete descriptor is built in temporary storage by compiled code.

The _event variable (EV)_ (if used) contains information about the
event that has been associated with the event I/O statement.
(For a description of the format, see "Event Table (EVTAB)" on
page 354). The implementation of event I/O is covered briefly
at the end of this chapter, and further in
Chapter 11, "Miscellaneous Library Subroutines and System
Interfaces" on page 230 for non-multitasking programs and
Chapter 14, "Multitasking" on page 307 for multitasking
programs.

The _abnormal locate return block_ is used only for LOCATE
statements.   It is the address of a block containing 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 data management routines
are shown in Figure 66 on page 169.

(3) Record descriptor
holds address and length
of record variable

(1) DCLCB
identifies file

(4) Key descriptor
holds address and length of key variable

(2) REQUEST CONTROL BLOCK
holds statement type

WRITE FILE (F) FROM (FS) KEYFROM(K);

EXECUTABLE INSTRUCTIONS (5) are a call to the PL/I library module IBMDRIO

completing and passing PARAMETER LIST (6) which holds addresses of 1, 2, 3 and 4.

---
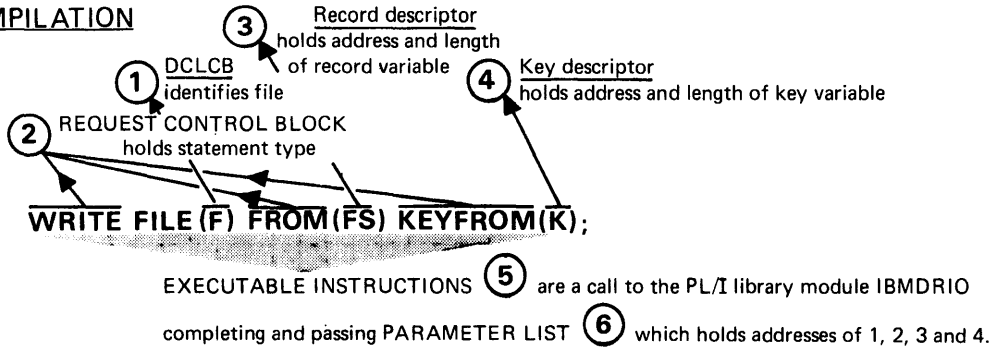
(1) DCLCB, set up from file declaration holds address of FCB via pseudo register vector.
(See file declaration).

(2) REQUEST CONTROL BLOCK holds record of statement type
000028 0880200091022001 CONSTANT

(3) RECORD DESCRIPTOR holds address and length of record, set up as far as possible during compilation, completed during execution. For statement above set up in temporary storage during prologue code

(4) KEY DESCRIPTOR holds address and length of key, set up as far as possible during compilation, but, for this statement, completely built by compiled code in temporary storage (see 5).

(5) Executable instruction

```
* STATEMENT NUMBER 4
000092  41 90 D 0B8    LA    9,184(0,13)    Pick up address record descriptor
000096  50 90 3 084    ST    9,132(0,3)     Place in parameter list
00009A  41 90 D 0B0    LA    9,176(0,13)    Pick up address key descriptor
00009E  50 90 3 088    ST    9,136(0,3)     Place in parameter list
0000A2  41 10 3 07C    LA    1,124(0,3)     Point R1 at parameter list
0000A6  58 F0 3 014    L     15,A..IBMBRIOA
0000AA  05 EF          BALR  14,15          Call IBMBRIO
```

Note: For this statement the record and key descriptors were set up in temporary storage during prologue code.

(6) PARAMETER LIST passed to IBMBRIO

```
00007C  00000000    A..DCLCB          Filled in by linkage editor
000080  00000028    A..CONSTANT       Request control block
000084  00000000    A..RD             (Record descriptor)
000088  00000000    A..KD             (Key descriptor (built during execution))
00008C  00000000    A..NULL ARGUMENT
000090  80000000    A..NULL ARGUMENT
```

Figure 66 (Part 1 of 2). Handling a Transmission Statement

EXECUTION OF TRANSMISSION STATEMENT

Call from compiled code   Return to compiled code

```
                   |                                    A
                   V                                    |
  +----------------------------------+                  |
  |                                  |                  |
  |             IBMBRIO              |                  |
  |                                  |                  |
  | (Resident library interface module) |               |
  | Loads parameters into registers. |                  |
  | Calls PL/I transient library     |                  |
  | transmitter whose address is placed |               |
  | in the FCB during the execution  |                  |
  | of the OPEN statement.           |                  |
  |                                  |                  |
  +----------------------------------+                  |
                   |                                    |
                   V                                    |
  +----------------------------------+                  |
  |                                  |                  |
  |         PL/I TRANSMITTER         |                  |
  |                                  |                  |
  | (Transient library)              |                  |
  | Calls data management.           |------------------+
  | Checks for errors and moves      |
  | record and key if necessary      |
  |                                  |
  +----------------------------------+
             |        A
             V        |
  +----------------------------------+
  |                                  |
  |         DATA MANAGEMENT          |
  |                                  |
  |    Handle the transfer of data   |
  |                                  |
  +----------------------------------+
```

Figure 66 (Part 2 of 2).   Handling a Transmission Statement

Execution

Compiled code calls the transmitter interface module, IBMBRIO, passing to it the parameter list shown above under "Compiler Output."

The interface module, IBMBRIO, first acquires a DSA, which is used by IBMBRIO and by the transmitter.   It then initializes the registers for the transmitter, and executes the TM instruction in the request control block (RCB).   This instruction tests a set of flags that are addressed by a pseudo-register offset contained in the DCLCB.   The contents of the pseudo-register offset depends on whether the file is open.   If the file is not open, it is opened, and return is made to this point to continue the statement.   (See "Implicit Open for Library-Call I/O" on page 176, for further discussion of this topic.)

When the file is open, the TM instruction tests the validity
flags in the FCB. This establishes the validity of the
statement. If the statement is not valid, a branch is made to
the address held in the word in the FCB following the statement
validity flags. This address is an entry point in IBMBRIO that
calls the error handling module, IBMBERR, with an error code
indicating an invalid statement.

If the statement is valid, a branch is made to the transmitter
whose address is held in the FCB.

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 the data management macro instruction
has been executed, 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 separate DSAs, but use the
DSA acquired by IBMBRIO.

If the statement is valid, control is returned to compiled code.
The situation when an error has been detected is described later
in this chapter under the heading "Error Conditions in
Transmission Statements."

In certain conditions, data management will require a parameter
list known as the data event control block (DECB). The PL/I
library routines include this block in a PL/I control block
known as the input/output control block (IOCB). A number of
IOCBs may be used. The number depends on the file type, and on
the NCP subparameter in the DD statement or NCP option in the
ENVIRONMENT attribute. Depending on the file type, IOCBs may be
generated during the execution of the open statement, or by the
transmitters when they are required.

The format of the IOCB is shown in "Input/Output Control Block
(IOCB)" on page 374. The format of the DECB and a further
description of its use is given in the publications OS/VS2 MVS
Data Management Macro Instructions, or in MVS/Extended
Architecture Data Management Macro Instructions. IOCBs are
further described in the section "EVENT Option," below.

EVENT OPTION

When the EVENT option is used, transmission statements are
always handled by library call. The compiler generates a call
to IBMBRIO in the usual manner, except that the address of an
event variable is passed in the parameter list.

The associated WAIT statement is compiled as a call to one of
the library wait modules. The module called depends on whether
or not the program is multitasking. The execution of an I/O
statement with the EVENT option and its associated WAIT
statement is shown in Figure 67 on page 172.

```
P : PROC;
    .
    .
READ....EVENT (E);
    .<
    .
    .                        ┌───────────────────────────────┐
    .                        │          IBMBRIO              │
    .                        └───────────────────────────────┘
    .
    .                        ┌───────────────────────────────┐
    .                        │       PL/I TRANSMITTER        │
    .                        │                               │
    .                        │  ISSUE DATA MANAGEMENT-        │
    .                        │  MACRO                         │
    .<                       │  RETURN IF EVENT I/O           │
    .                        │  ISSUE CHECK MACRO             │
    .                        │                               │
    .                        │  TEST FOR ERRORS              │
    .                        │  IF NONE RETURN TO WAIT       │
    .                        │  MODULE                        │
    .                        └───────────────────────────────┘
    .
WAIT (E);
    .
    .                        ┌───────────────────────────────┐
    .<                       │  WAIT MODULE                  │
    .                        │                               │
    .                        │  IF EVENT I/O CALL IBMBRIO    │
    .                        │                               │
    .                        │  RETURN IF NO MORE            │
    .                        │  EVENTS TO WAIT ON            │
    .                        └───────────────────────────────┘
    .
END P;                Key
```

Key
```
────────>──────── READ EVENT statement
─ ─ ─>─ ─ ─ WAIT statement
.............. Further PL/I statements
```
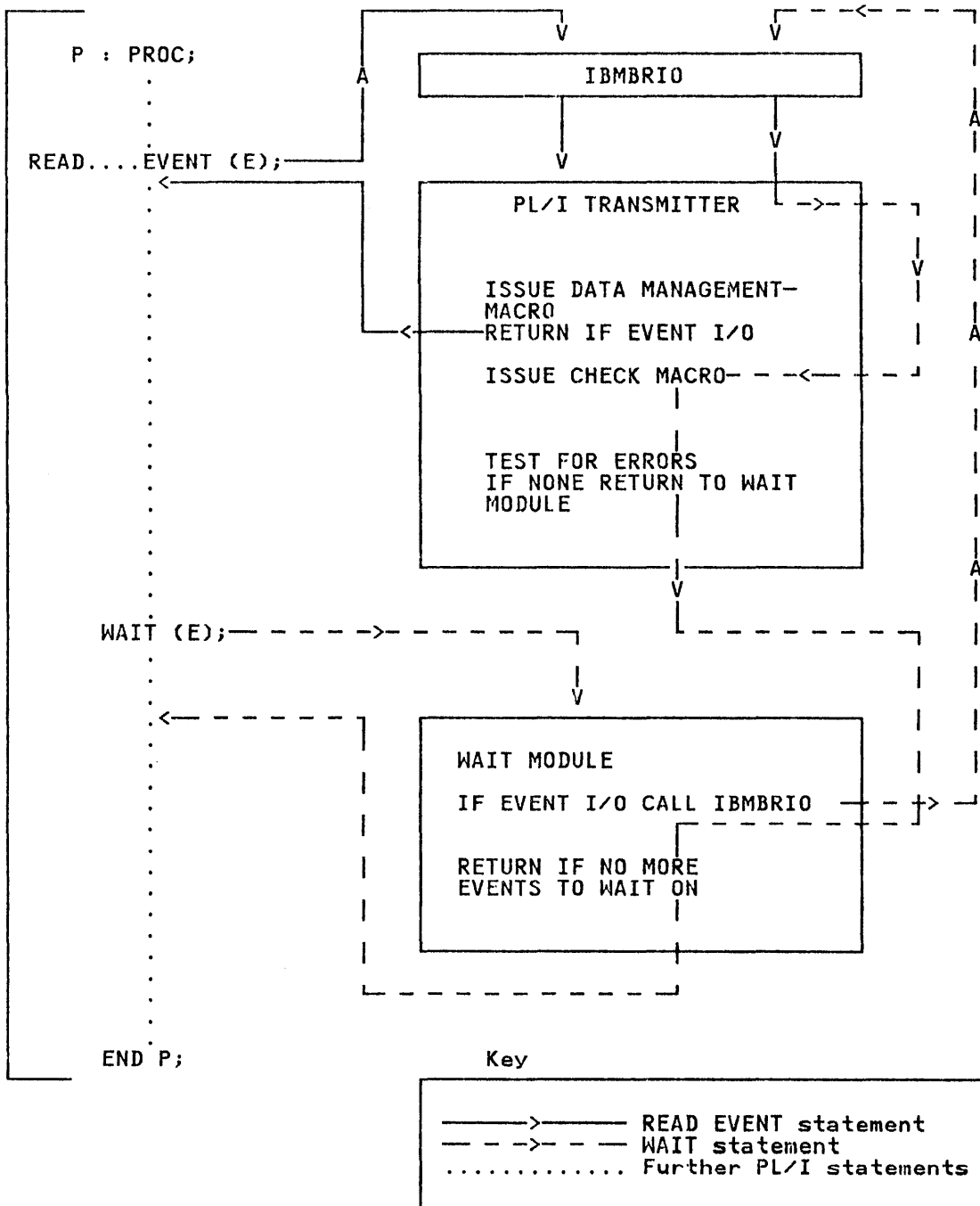
Figure 67.   Handling the EVENT Option

Execution

The principle used in event I/O is that the PL/I transmitter
returns to compiled code as soon as the data management macro
instruction has initiated the I/O.

When I/O with the EVENT option is being executed, the event
variable associated with the event is set active and flagged to
indicate that the event is an I/O event. When the WAIT
statement is reached, the library wait module is entered. When
the event is an I/O event, the PL/I library wait routine passes

control to IBMBRIO. From information in the event variable,
IBMBRIO locates the I/O operation associated with the event, and
calls the transmitter. The transmitter then issues a CHECK
macro instruction, and waits until the operation is complete.
When control returns after the CHECK macro instruction, the
transmitter assigns the transmitted data, and either returns to
the wait module, or, if any errors are detected, enters one of
the error routines. (For further details, see "Error Conditions
in Transmission Statements" on page 176.)

When the transmitter assigns the data, it is necessary for the
address and length of the record variable, and certain other
information, to be available. This information is retained in
the input/output control block (IOCB).

Use of the IOCB

The IOCB is chained to the event variable so that the I/O
routines can access the statement when control is returned to
them during execution of the WAIT statement.

To associate the PL/I statement with the data management
operation, the DECB for the operation is included in the IOCB.
(The DECB is a record held by the data management routines so
that the operation can be posted complete.)

For certain types of PL/I files, the IOCB also contains the data
management buffer to or from which the transmission will be
made.

Allocation of IOCBs

For direct access files, IOCBs are allocated as they are
required by the transmitter.

For sequential access files, the IOCBs are generated by the open
routines. The number of IOCBs requested corresponds to the
number specified in the NCP subparameter or option.

IOCBs and Dummy Records

In event I/O, the existence of a dummy record may not be
discovered until after a read has commenced on the record
following the dummy. When this happens, the DECB and IOCB
pointers are reset appropriately.

Raising Conditions in Event I/O

Because the CHECK macro instruction is not issued until the WAIT
statement is executed, PL/I conditions raised in event I/O are
handled during execution of the WAIT statement. The
implications of this are discussed in the section on the WAIT
statement in Chapter 11, "Miscellaneous Library Subroutines and
System Interfaces" on page 230 for non-multitasking programs,
and Chapter 14, "Multitasking" on page 307 for multitasking
programs.

Exclusive I/O

In exclusive I/O, records are protected from simultaneous
updates from different tasks by use of the ENQ and DEQ macro
instructions.

When a READ statement for an exclusive file is being executed,
an ENQ macro instruction is issued. Unless NOLOCK is specified,
the DEQ macro instruction is not issued until a REWRITE, DELETE,
or UNLOCK statement is executed. For unblocked records, the ENQ
and DEQ instructions are issued on one record only. For blocked
records, they are issued on the data set.

Eight PL/I transmitter modules are used to handle exclusive files. They are shown in Figure 60 on page 157. The ENQ and DEQ macro instructions are issued by calling the resident library routine IBMBPDQ, which is addressed from the TCA.

The protection of the data set depends on all files that access the data set having the EXCLUSIVE attribute. If the data set is accessed by a file that does not have the EXCLUSIVE attribute, the data set will not be protected.

For VSAM files the EXCLUSIVE attribute is ignored and the NOLOCK option and UNLOCK statement will have no effect (except that for UNLOCK, the key specification is checked.) Data set protection is provided by VSAM itself.

CLOSE STATEMENTS AND IMPLICIT CLOSE

Compiler Output

For CLOSE statements, the compiler generates a call to the appropriate entry point of the open/close bootstrap module, passing it the addresses of the DCLCB and ENVB for the file.

No compiler action is taken for implicit close.

Execution

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 library routines. The bootstrap module IBMBOCL is called either by compiled code, or, during program termination, by the termination routine, IBMBPIT or IBMTPJR for multitasking. It loads and calls the transient close routine, IBMBOCA.

The bootstrap routine IBMBOCL is passed a parameter list containing the addresses of the DCLCBs and ENVBs for the files that require closing. IBMBOCA then closes these files. This may involve completing I/O operations, and hence calling the transmitter. After handling any necessary transmission, IBMBOCA disassociates the file from the data set.

The ENVB is required if the LEAVE or REREAD option is in effect.

For implicit closing, the chain of open files starting in the TCA is scanned to determine which files must be closed. The addresses of the FCBs of these files are then passed to the close routine.

For an explicit close, it is necessary to set the address in the pseudo-register vector to point, once more, to the dummy FCB. This allows implicit opening to be handled should the file be opened again. (See "Implicit Open for Library-Call I/O" on page 176 for further details.)

When IBMBOCA has finished, it returns control (via IBMBOCL) either to compiled code (for an explicit close statement) or to the termination routine (for the end of the program). The code and control blocks generated for a CLOSE statement are summarized in Figure 68 on page 175.

① DCLCB identifies file to be closed

CLOSE FILE (F2)

② Executable instructions consist of a call to the open/close bootstrap module passing parameter list ③

---

① DCLCB set up for file declaration see figure 8.5

② Executable instructions

```
* STATEMENT NUMBER 5
0000AC 41 10 3 094      LA    1,148(0,3)         Place address DCLCB in p-list
0000B0 58 F0 3 010      L     15,A. .IBMBOCLC  ⎫
0000B4 05 EF            BALR  14,15            ⎬ Call open/close toolstrap
                                               ⎭
```

③ Parameter list

```
000094 00000044   A. .CONSTANT        Address of constant showing number of files to be closed
000098 00000000   A. .DCLCB           Address DCLCB
00009C 80000000   A. .NULL ARGUMENT   Used for disposition options, flagged in first bit to indicate last argument
```

CLOSE FILE (F1);

COMPILATION

```
L     7,F0              Pass address of constant with number of files to be closed
ST    7,2528(0,3)       Pass address of DCLCB of file
LA    1,2524(0,3)       Point R1 at parameter list
L     15,A. .IBMBOCLC   Branch to open/close bootstrap
BALR  14,15
```

EXECUTION

Call from compiled code   Return to compiled code

> IBMBOCL
> Entry point C
>
> Resident library open/close bootstrap
> routine. Calls the close routine

> IBMBOCA
>
> Transient library close routine. Calls
> transmitter to complete I/O if necessary.
> Calls data management to close the data
> set. Removes FCB from Open File
> Chain. Restores PRV offset to point to
> dummy FCB.

> DATA MANAGEMENT
>
> Disassociates file from data set.

> PL/I TRANSMITTER
>
> Transient library routine. Calls
> data management to complete I/O

Figure 68.   The Execution of an Explicit CLOSE Statement

IMPLICIT OPEN FOR LIBRARY-CALL I/O

Compiler Output

There is no compiler output for an implicit open, because it is not always possible to predict which transmission statements will cause implicit opening of a file.

Execution

Implicit opening is handled by manipulation of addresses (see Figure 69 on page 177).

When IBMBRIO is called for a transmission statement, it executes a test-under-mask (TM) instruction against a set of flags held at an offset from the address held in the pseudo-register vector. The address held in the pseudo-register vector depends on whether the file is open. If the file is open, the pseudo-register offset contains the address of the FCB for the file. If the file is not open, the pseudo-register offset contains the address of a dummy FCB in the program management area.

The address is set during program initialization to point to the dummy FCB, and is reset to the dummy FCB whenever a file is closed.

The first word in the dummy FCB is a set of statement validity flags. These are all set to zero. Consequently any TM instruction executed by IBMBRIO will give a negative result. The second word of the dummy FCB is the address of an entry point in the open/close bootstrap module. If the TM instruction yields a negative result, IBMBRIO branches to the address held immediately after the statement validity flags. Consequently when an attempt is made to execute a transmission statement on a file that is not open, control passes automatically to the open routines.

The open routines open the file, and set up an FCB and DCB for the file. The address of the FCB is placed in the pseudo-register offset, and execution of the statement is reattempted by branching once more to IBMBRIO.

ERROR CONDITIONS IN TRANSMISSION STATEMENTS

To provide PL/I error handling facilities with the minimum possible overhead to error-free programs, transient-library modules are used. These are not loaded unless an error occurs. Two modules are available for every file type except VSAM:

1. The ENDFILE routine, IBMBREF, which can deal only with the ENDFILE condition.

2. A general error module capable of handling all conditions that may arise, including ENDFILE, but loaded only if the TRANSMIT, RECORD, KEY, or ERROR condition occurs. (See Figure 70 on page 178.)
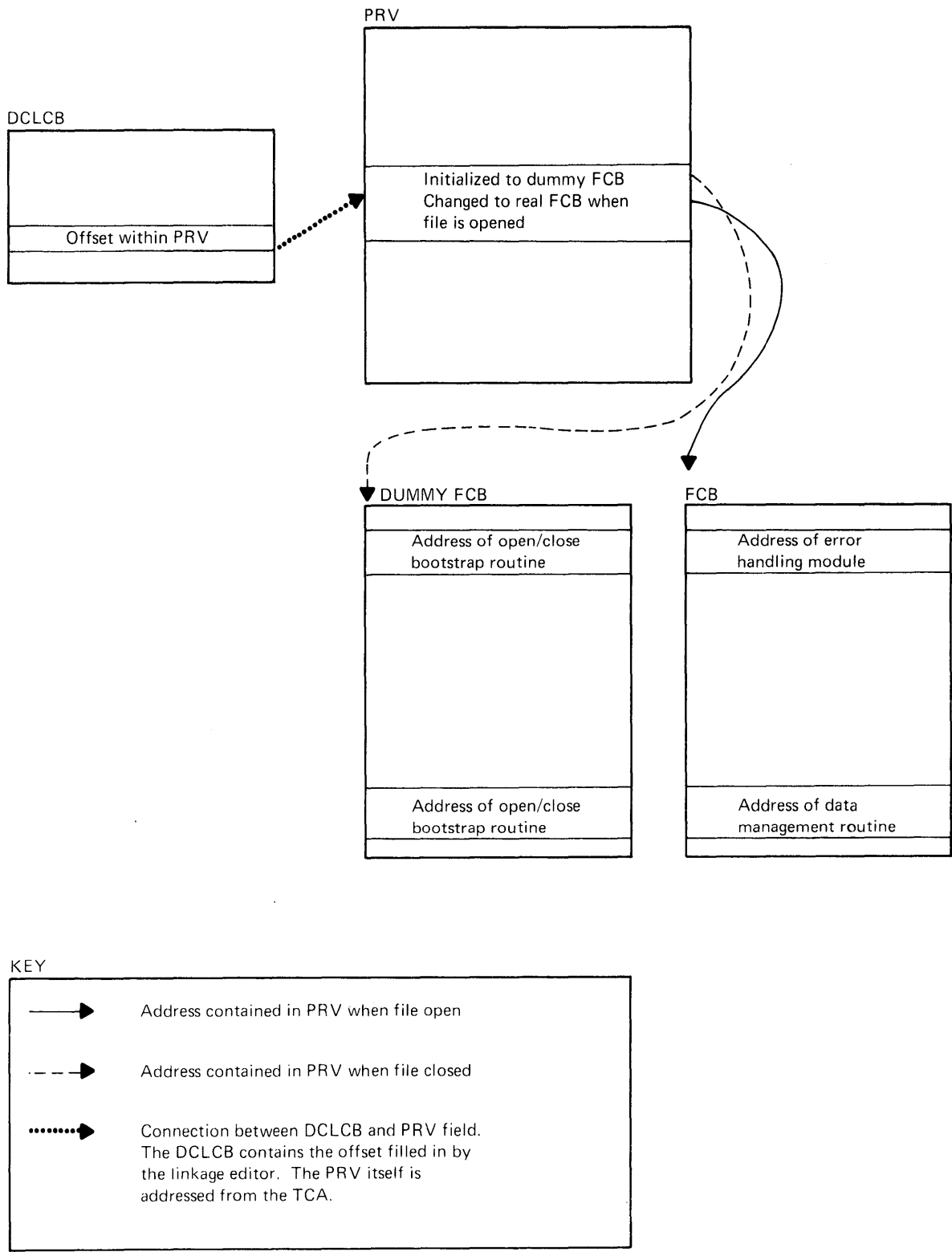
Figure 69.   The Addressing Mechanism Used during Implicit Open

Record I/O Error Module	File Types
IBMBREA	Consecutive buffered
IBMBREB	Indexed
IBMBREC	Regional, consecutive unbuffered, and transient
IBMBREE	VSAM
ENDFILE Module IBMBREF	All SEQUENTIAL/INPUT/ UPDATE file types (excluding VSAM)

Figure 70. Record I/O Error Modules

This method is used because the short ENDFILE module gives faster execution to those programs that use the ENDFILE condition to handle program flow. The transient error modules for all file types are identified by the six letters IBMBRE followed by a further single character (see Figure 70).

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. The transmission error routine has been nominated in the SYNAD exit address placed in the DCB by the OPEN routines. 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. The address may be any one of the following:

* The address of IBMBREF, the ENDFILE module.

* The address of the general error module for the file type.

* The address of a bootstrap routine, IBMBRIOB. This routine constructs the name of an error module by taking the skeleton IBMBRExA and replacing the "x" by the letter in the single character field "FEFT" in the FCB. IBMBRIO then loads this error module, places the address of the module in FERM, and branches to the module.

So, 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:

* 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.

* The character "F," indicating the name of the ENDFILE module. The contents 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 IBMBRIOB, and the address placed in FERM, depend on whether end-of-file or another error is the first to occur on the file.

Contents FEFT

Initialized by open routine with character "A", "B", "C", "E" indicating general error support module.
Altered by end of file routines in transmitter to character "F" indicating ENDFILE module

IBMBRIO
(entry point B)
Loads and calls module indicated in "FEFT" and places its address in FERM.

FCB

FEFT
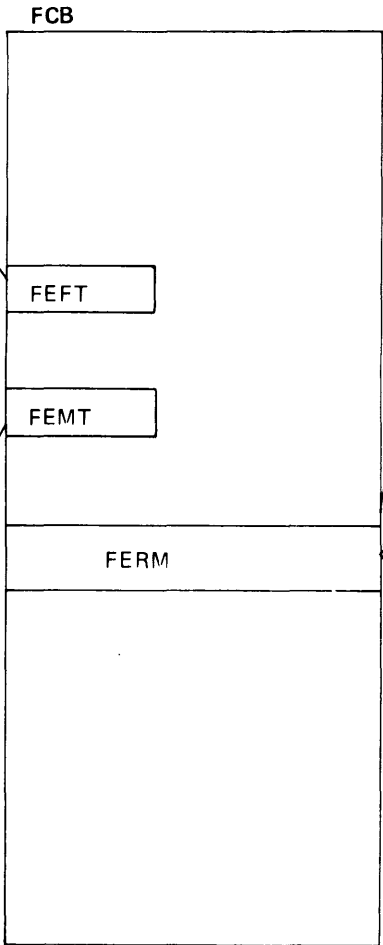
FEMT

FERM

IBMBREF
Endfile module
If ENDFILE :
Calls error handler
If other error :
Loads and calls error module indicated in "FEMT". Placing address in FERM

Contents FEMT
Always contains character indicating general error support module

IBMBRE/A/B/C/E
General error support modules.

Handle all errors including ENDFILE

Key

———————	If no errors have occurred.
●●●●●●●	If 1st. error was ENDFILE and no other errors occurred.
— — — — —	If non-ENDFILE errors have occurred.
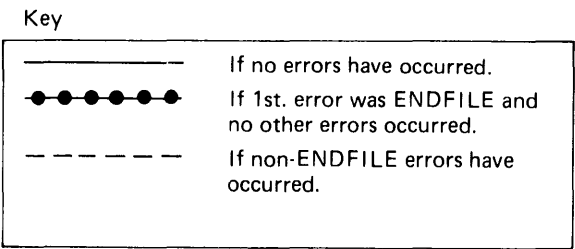
Figure 71. The Fields Used in Record I/O Error Handling

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 constructs the name of the general error module in a similar manner to that used by IBMBRIOB, described above. However, the sixth letter of the name is taken from a field in the FCB called "FEMT". 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 by way of the bootstrap routine in IBMBRIO.

General Error Routines (Transient)

The general error routines set up a parameter list and the relevant built-in function values in the ONCA (described in Chapter 7). They then call the resident error handler IBMBERR 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 by calling IBMBERR with the appropriate error code. 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.

TRANSMIT Condition

For certain file types, when a permanent transmission error occurs, action must be taken to prevent subsequent issuing of data management macro instructions. To achieve this, addresses are manipulated so that, instead of IBMBRIO calling the transmitter by its primary entry point, it calls an error routine within the transmitter, which in turn calls the error handler to raise the TRANSMIT condition.

IN-LINE I/O STATEMENTS

Most transmission statements on buffered consecutive files are implemented by short in-line calls to the data management routines (see Figure 74 on page 184 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 are always executed by library calls.

Control Blocks

For in-line I/O statements, the only control blocks that are set up are the FCB and DCB. The request control block, record descriptor, and key descriptor are not required as they are merely parameters for full library subroutines.

Executable Instructions

For in-line I/O, a call is made to a special entry in a transmitter. In an MVS/XA environment, this transmitter provides the correct addressing mode and directly calls the data management routine via the address held in the FCB for output files, and in the DCB for input files. In addition to calling the data management 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.

For U-format and V-format records on output files, compiled code does not call data management direct. Instead a call is made to another short call within the PL/I transmitters. These routines are addressed through the field in the FCB that normally addresses the data management routines. This field is initialized by the open routines when U-format or V-format records are used on the file. The compiler can thus produce the same code for all record types.

For certain types of blocked file, deblocking is handled by compiled code. Fields in the DCB hold the address of the current record, the address of the end of the block, and the record length. Before a call is made to data management, a check is made to see whether the end of the block has been reached. This is done by adding the record length to the current record address. If the resultant address is the end of the block, a call is made to data management for a new block; otherwise, the new address can be taken as the start of the required record.

Error Conditions

If an error occurs during transmission, or if end-of-file is reached, the data management routines will branch to the ENDFILE or SYNAD routines that are held in the PL/I transmitter. (The PL/I transmitter is _always_ loaded by the open routines.) The ENDFILE and SYNAD routines set an error flag in the FCB, and return to compiled code, normally via the data management 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 72 on page 182.

Implicit Open for In-Line Calls

Implicit opening for in-line calls is handled in a similar way to that used for library calls.

The field that, in a normal FCB, points to the data management transmitter, in the dummy FCB points to the open/close bootstrap routine, IBMBOCL (see Figure 69 on page 177). This results in a branch being made to the OPEN routines when an attempt is made to access a file that is not open. When the open routines have been executed, the address in the pseudo-register vector is altered to point to the FCB that has been created for the file.

If the file is successfully opened, a test is made to see whether the entry to IBMBOCL was for an in-line call and, if it was, control is passed to the data management address held in the DCB. This causes the data management module to be entered and a return made to compiled code.

```
            SOURCE STATEMENTS

  1          TOTAL: PROC OPTIONS(MAIN);
  2   1             DCL LINE FILE RECORD INPUT
                        ENV(FB,RECSIZE(80),BLKSIZE(400),TOTAL);
  3   1             DCL CARD CHAR(80);
  4   1             READ FILE(LINE) INTO(CARD);
  5   1             END TOTAL;
```

```
* STATEMENT NUMBER  4
00005E  18 72                    LR   7,2              Save program base
000060  58 F0 3 024              L    15,36(0,3)       Load R15 address of DCLCB
000064  18 BF                    LR   11,15            Load R11 DCLCB
000066  58 10 C 004              L    1,4(0,12)        Load R1 PRV
00006A  5A 10 B 000              A    1,0(0,11)        Add PRV offset in DCLCB to
                                                          address in R1
00006E  58 20 1 000              L    2,0(0,1)         Point R2 at FCB
000072  58 10 2 014              L    1,20(0,2)        Point R1 at DCB
000076  18 81                    LR   8,1              Load address of DCB
000078  BF 17 8 04D              ICM  1,7,77(8)        Get last record address
00007C  4A 10 8 052              AH   1,82(0,8)        Add logical record length
                                                          to access required record
000080  59 10 8 048              C    1,72(0,8)        Compare with end of buffer
000084  47 40 7 03A              BL   CL.2             Branch around Library call
000088  18 18                    LR   1,8              Restore DCB address if a new
                                                          buffer is required
00008A  41 80 3 028              LA   8,40(0,3)        Pass abnormal return address
                                                          (CL.3) in R8 for error
                                                          handling
00008E  58 F0 2 01C              L    15,28(0,2)       Get short transmitter
000092  05 EF                    BALR 14,15            Branch and link to data
                                                          management routine
000094  47 F0 7 03E              B    CL.4             Don't need next instruction
000098                   CL.2    EQU  *               Label branched to, if no data
                                                          management call
000098  BE 17 8 04D              STCM 1,7,77(8)        Save record address
00009C                   CL.4    EQU  *
00009C  D2 4F D 0B8 1 000        MVC  CARD(80),0(1)    Move record into record
                                                          variable
0000A2                   CL.3    EQU  *
0000A2  91 C0 2 02C              TM   44(2),X'C0'      Test for errors
0000A6  47 80 7 052              BZ   CL.5             Branch if no errors
0000AA  58 F0 3 01C              L    15,A..IBMBRIOD   If errors, call error
                                                          bootstrap routine
0000AE  05 EF                    BALR 14,15
0000B0                   CL.5    EQU  *
0000B0  18 27                    LR   2,7              Restore Program Base
```

Figure 72. In-Line I/O Transmission Statement

A further problem arises over deblocking. For certain blocked
files, before data management is called, a test is made to see
whether the end of the block has been reached. For such files,
values are placed in the dummy FCB that ensure that if the test
for end-of-block is made before the file has been opened, the
test will reveal an apparent end-of-block. A branch will
therefore be made to the transmitter field in the dummy FCB, and
control will pass to the open/close bootstrap routine.

Figure 73. Overview of Record I/O

EXECUTION

| SOURCE PROGRAM | COMPILATION | COMPILED CODE | LIBRARY AND DATA MANAGEMENT MODULES | COMPILED CODE |

Open statement

Generate code to call open bootstrap

Call IBMBOCL

EXPLICIT OPEN   IMPLICIT OPEN

Transmission I/O statement

IBMBOCL
Pass control block addresses to IBMBOPA

Validity check possible?

NO

YES

Generate in-line code

In-line

IBMBOPA, etc.
Associate file with data set Load transmitter

Valid statement?

YES

In-line or library?

Library calls   In-line calls

NO

LIBRARY

Set up RCB for Execution-Time test

Set up RCB with code to branch found Execution-Time test

IBMBRIO

Branch to ERROR handler (IBMBERRA) or open bootstrap if file not open

NO

TRANSMITTER

DATA MANAGEMENT

Call data management routine

Carry out I/O

Call data management

Valid statement?

YES

TM Instruction

Any Errors?

NO

Generate call to IBMBRIO

Call IBMBRIO

Execute Instruction in RCB

Branch instruction

Error routines set flags indicating error has occurred

YES

NO

Move record key etc. Check for record & key errors

Move record key etc. Check for record & key condition

Key

———— Path using library calls
·········· Path for in-line I/O
•—•—•— Common path, in-line/library
— — — Path for implicit open

Any conditions to raise ?

YES

Record I/O Error module
Call IBMBERR

Any conditions to raise ?

YES

NO

CONTINUE

NO

CONTINUE

File type: Consecutive buffered (TOTAL option used)

Statement	Record Variable Requirements	ENVIRONMENT Option Requirements
Record type: F,FB		
READ SET	None	None
READ INTO	Length known at compile time (max. length if a varying string or area[1])	RECSIZE known at compile time SCALARVARYING option if varying string
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[1]		RECSIZE known at compile time
LOCATE A	Length known at compile time (max. length if varying string or area[1])	RECSIZE known at compile time SCALARVARYING if varying string

Statement	Record Variable Requirements	ENVIRONMENT Option Requirements
Record type: U,V,VB		
READ SET	None	Not BACKWARDS
READ INTO	Length known at compile time (max. length if a varying string or area[1])	RECSIZE known at compile time SCALARVARYING if varying string
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[1]		RECSIZE known at compile time
LOCATE	Length known at compile time (max. length if varying string or area[1])	RECSIZE known at compile time SCALARVARYING if varying string

Figure 74.   Conditions under Which I/O Statements Are Handled In-Line

[1]   Including structures whose last element is an unsubscripted area.

Notes to Figure 74: All statements must be found to be valid during compilation. File parameters or file variables are <u>never</u> handled by in-line code.

BLKSIZE may be specified instead of RECSIZE for F, V, and U formats (but not FB, VB).

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 75 on page 186. 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 routines of OS data management. These routines are called by the PL/I transient library transmitters in the same way as that used in library-called record I/O. The movement between the buffer and the PL/I variables is generally handled by the PL/I conversion routines. The transmitters and the conversion routines are called by director routines. These routines determine which modules are required, and when they are needed.

Data items transmitted by stream I/O are not affected by record boundaries (see Figure 76 on page 187). There may be any number of data items in a record, and an item may span any number of records. Because the data management 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.

PL/I Statement: GET LIST(I);

```
                        ┌─────────────────────────────┐
                        │ PL/I transmitter modules    │
                        │ call LIOCS routines to move  │
                        │ the data between the external│◄─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─┐
                        │ medium and the data manage-  │                           │
                        │ ment buffer.                 │                           │
                        └─────────────┬───────────────┘                           │
                                      │                                            │
 External medium File SYSIN           │         Data Management buffer             │
┌─────────────────────────────┐       ▼        ┌───────────────────────────────┐   │
│ 8    9    10    1           ╲        Stage 1 │ 8    9    10    1             │  ▲ │
│                              ╲──────────────►│                               │   │
└─────────────────────────────┘                └───────────────────────────────┘   │
                                                                                    │
                                                                                    │
                                                                                    │
                                                                                    │
                           Stage 2                                                  │
                        ┌─────────────────────────────┐  ┌───────────────────────────┐
                        │ Conversion routines or      │  │ Director routines control the│
                        │ compiled code convert       │  │ process, calling necessary │
                        │ data and move to variable.  │  │ conversion and transmitter │
                        │                             │  │ modules when required.     │
┌─────────────────────┐ │                             │◄─│                           │
│ 0000000000001000    │◄┤                             │  └───────────────────────────┘
└─────────────────────┘ └─────────────────────────────┘

Variable I (Fixed Binary 15,0)
(in main storage)
```

Figure 75. The Principles Used in Stream I/O

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 76. Record Boundaries Do Not Affect Stream I/O

Operations in a Stream I/O Statement

A stream I/O operation can involve any or all of the following operations:

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.

6. Enqueuing and dequeuing on SYSPRINT.

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/termination 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. The initialization/termination routine is also called when every PUT statement is completed, to dequeue on SYSPRINT and, for conversational files, to complete the output. The routine is also called on the completion of GET statements with the COPY option, to transmit the data to the copy file.

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 77 on page 189. The sequence is:

1. A call to the initialization module, to check statement validity, and to open the file if necessary.

2. A return to compiled code, to set up parameters for the director module.

3. A call to the director module to handle any conversion, transmission, and housekeeping problems that may be involved.

4. For PUT statements, a terminating call to the initialization/ termination routine to dequeue on SYSPRINT.
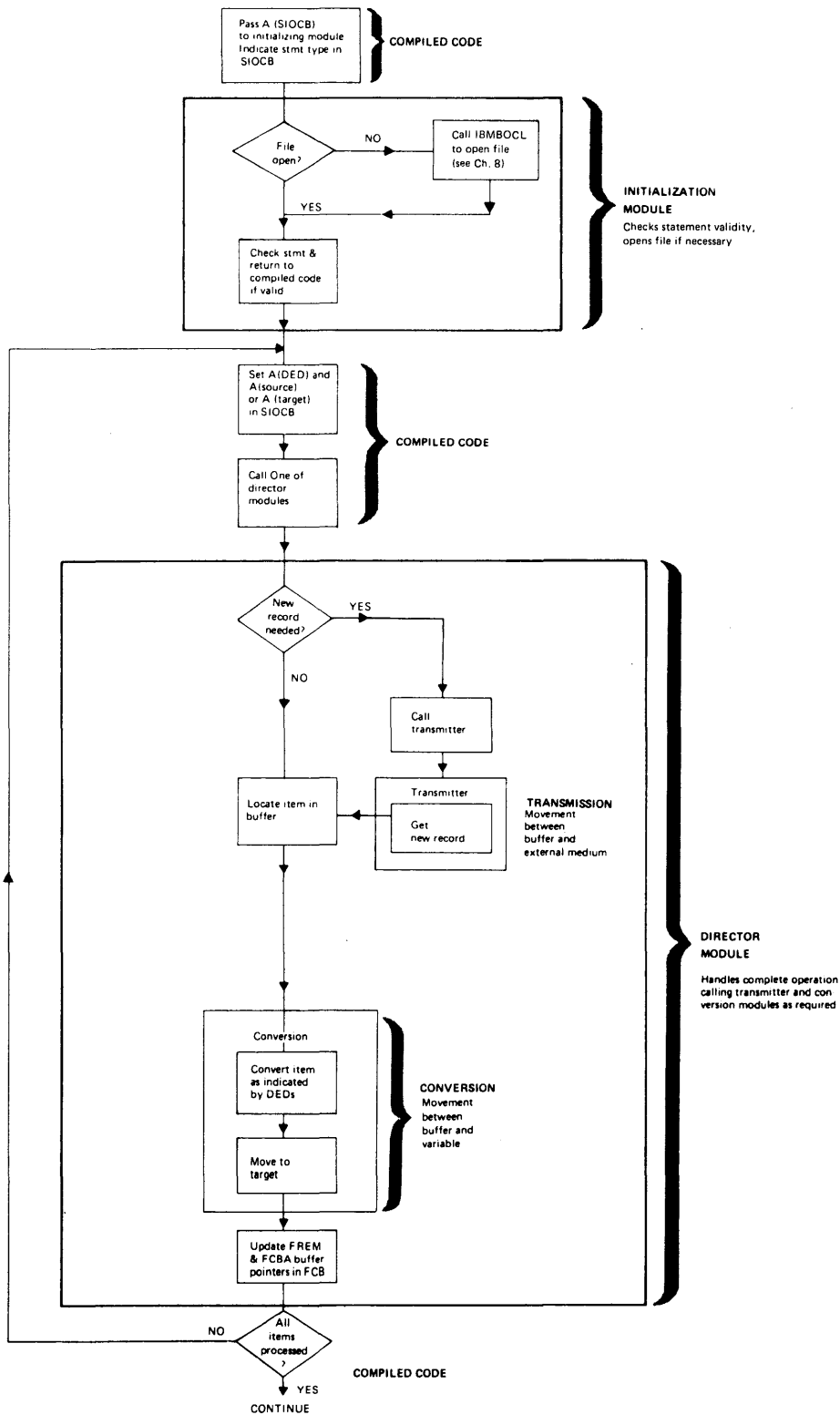
Figure 77.   Simplified Flow Diagram of a Stream I/O Statement

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 the 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 78. The SIOCB contains the addresses of the source and target (or their locators), and of the DEDs of the source and the target. The SIOCB is passed directly to the conversion routines. The first four words contain the parameters expected by the conversion routines.

Field	Contents
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 control block. (27 fullwords)

Figure 78. Stream I/O Control Block (SIOCB)

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, in certain cases, by the initialization module, or by the close module to complete transmission when the program is terminated.

As with record I/O, transmitters call data management modules. The PL/I transmitters contain the EODAD and SYNAD routines, which are entered when end-of-file or other errors are detected in the routines. Nine different transmitter modules are used in stream I/O; these include two for conversational files. The stream I/O transmitters are listed in "Transmitter Modules" on page 217.

Opening the File

The same basic method is used for opening the file as is used for record I/O. During compilation, a declare control block (DCLCB) and an environment control block (ENVB) are set up. An open control block (OCB) is also set up if any environment options are declared in the OPEN statement. At open time, the information addressed from the DCLCB, ENVB, and the OCB (if any) is merged with any information in the DD statement, and an FCB is set up. The PL/I transmitter is loaded, and its address placed in the FCB. A DCB, addressed from the FCB, is set up. The DCB contains the address of the data management transmitter. Finally, the address of the FCB is placed in the pseudo-register vector.

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 IBMBOCL.

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. The format of a stream I/O FCB is shown in "Stream I/O Control Block (SIOCB)" on page 400.

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:

FCBA   Points at the position reached in current record.

FREM   Number of unused bytes remaining in the record.

FCBA points to the position reached in the record and enables the director routines to identify from where the next input item must be read, 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 79 on page 192 illustrates the use of FCBA and FREM.

Enqueuing and Dequeuing on SYSPRINT

Because SYSPRINT is used as the standard file for error messages, it is necessary for output to SYSPRINT to be enqueued. This prevents error messages from one task in a PL/I program interrupting other output to SYSPRINT from another task.

When SYSPRINT is used it is enqueued by the initialization routine. When any PUT statement is completed, regardless of the output file, a call is made to the initialization/termination routine. This routine then checks to see if SYSPRINT has been enqueued and, if it has, dequeues it by calling the DEQ routine.

PL/I STATEMENT:

GET FILE (SYSIN) LIST (A, B);

80 Byte record
In data management buffer

FCBA
Holds address reached

At start of first item   after processing first item   start of second item   after processing second item

| FREM VALUE | 80 | | FREM VALUE | 50 | | FREM VALUE | 41 | | FREM VALUE | 3 |
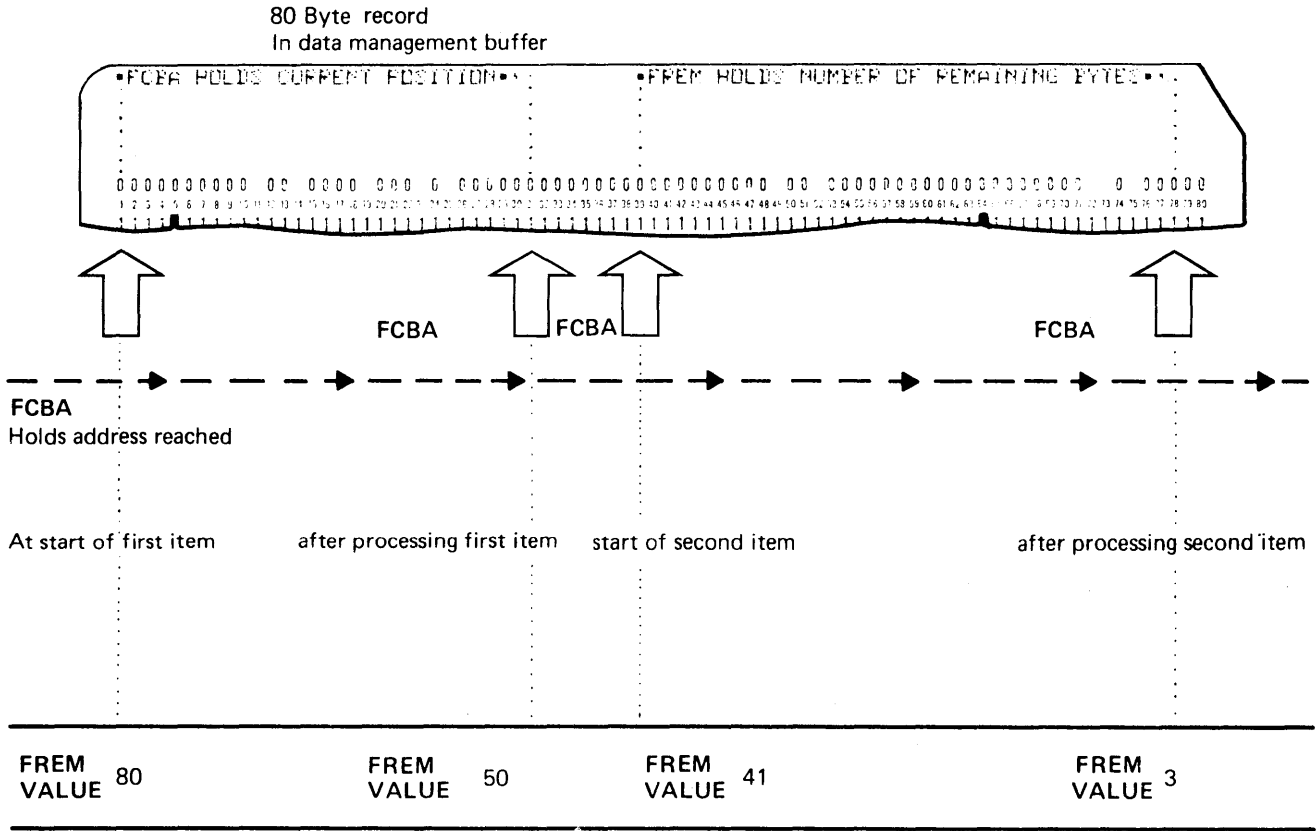
FREM holds number of remaining bytes

Figure 79.   The Use of FREM and FCBA in Recording Buffer Position

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, and determine which entry point of which conversion module is required. Each possible conversion has a unique entry point in one of the conversion routines. For stream I/O, the argument list passed is contained in the first four words of the SIOCB.

A number of conversion director modules are used exclusively by edit-directed stream I/O. These are called <u>external conversion directors</u>, and are listed in "External Conversation Director Modules" on page 218. Each module corresponds to a particular format of input or 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. A third possibility is that compiled code will 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 directly. 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 in handling the CONVERSION condition and other PL/I exceptional 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. A generalized impression is given in Figure 77 on page 189 and summarized above.

This chapter first covers the implementation of list-directed GET and PUT statements in some detail, and then highlights the differences for data-directed and edit-directed I/O.

LIST-DIRECTED GET AND PUT STATEMENTS

PUT LIST Statement

Implementation of a list-directed output statement is shown in Figure 80 on page 195. The process consists of five steps:

1. Compiled code calls the initialization routine, passing the address of the DCLCB and of the SIOCB. Flags indicating the statement type have been set in the SIOCB by compiled code.

2. The initialization routine, IBMBSIO, 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.

   IBMBSIO also handles any format options, by calling the formatting module IBMBSPL. 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 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 may be the default tab table in the library module IBMBSTAA. For non-print files, each item is followed by a single blank. PLITABS is addressed from the TCA.

   When the starting position for the item has been found, the director module determines whether there is enough space in the output buffer for the converted item. There may not be, for one of two reasons:

   a. The end of the buffer has been reached.

   b. The converted item will be too large to hold in the buffer.

   If the end of the buffer has been reached, the transmitter is called to acquire a new record. If the converted item will be too long to fit in the buffer, intermediate workspace will be needed.

   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—see Chapter 6). The converted item is then placed in the VDA. As much of the data as will fit is moved from the VDA into the data management buffer, and a new record is acquired by a call to the output transmitter. The new record is then filled from the VDA. This process is continued until the complete item has been moved into buffers. The buffer pointers FREM and FCBA 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.

5. The statement is completed by a call to the initialization/ termination routine. This checks to see whether SYSPRINT has been used and, if so, dequeues on SYSPRINT. For conversational files, it also calls the transmitter to transmit any information that is still held in the buffer.

The object code produced for a PUT LIST statement is shown in Figure 81 on page 197.

PUT LIST (A)

FLOW DIAGRAM		COMPILED CODE & NOTES

Step 1
Compiled code

```
LA    7,200(0,13)        Load address SIOCB
ST    7,76(0,3)          Place in parameter list
OI    76(3),X'80'        Mark end of parameter list
LA    1,200(0,13)        Saves SIOCB in a temporary
ST    1,192(0,13)            pointer
MVI   217(13),X'40'      Set LIST OUTPUT flag
LA    1,72(0,3)          Point R1 at parameter list
L     15,A. .IBMBSIOA    Call stream output
BALR  14,15                  initializer
```

Place address SIOCB in parameter list

Call initializing module

Step 2
Initializing routine
IBMBSIO

The initialization routine is passed the address of the FCB and the address of the SIOCB.

Is File open ? — NO → Call IBMBOCL to open file & load transmitter

YES

It opens the file if necessary and acquires the first record for print files. If the statement is invalid it calls the error handler. If the statement is valid it places the addresses of the ONCA and the FCB in the SIOCB and returns to compiled code.

Is statement valid ? — NO → Call error handler

YES

Set FCB & ONCA address in SIOCB

Figure 80 (Part 1 of 2). Flow of Control through a PUT LIST Statement

Step 3
Compiled code

Point R1
at
SIOCB

Put address
of DED &
source variable
in SIOCB

LA	14,A	Get the address of A
LA	15,DED. .A	Get the address of DED. .A
L	1,192(0,13)	Reloads R1
STM	14,15,0(1)	Put addresses of A and DED. .A in SIOCB
L	1,192(0,13)	Restore SIOCB
L	15,A. .IBMBSIOT	Call termination
BALR	14,15	routine

Step 4
Director module
IBMBSLO

The director module calls the transmitter and
conversion modules when required and handles
any housekeeping problems.

Reached
end of current
record
?
— YES → Call transistor
for new record

NO

Before calling the conversion module it completes
the SIOCB with the address of the target locator
and the address of the target DED.
The target for the conversion is either the data
management buffer or a VDA acquired for
intermediate workspace.

Will
converted
item span record
boundary
?
— YES → Get VDA & set
as target for
conversion

NO

Set target
address in
SIOCB
Call conversion
module

VDA
used as
target
?
— YES → Fill record
from VDA -
call transmitter

NO

Update FCBA
& FREM ← YES — VDA
exhausted? — NO →

Continue as
from Step 3
until state-
ment complete
When complete
call termination
routine

Compiled code

If the statement is complete compiled code continues
with the next statement. If the statement is not complete
compiled code places new data in the SIOCB and once
more calls the director module.
When statement complete make terminating
call to dequeue on SYSPRINT
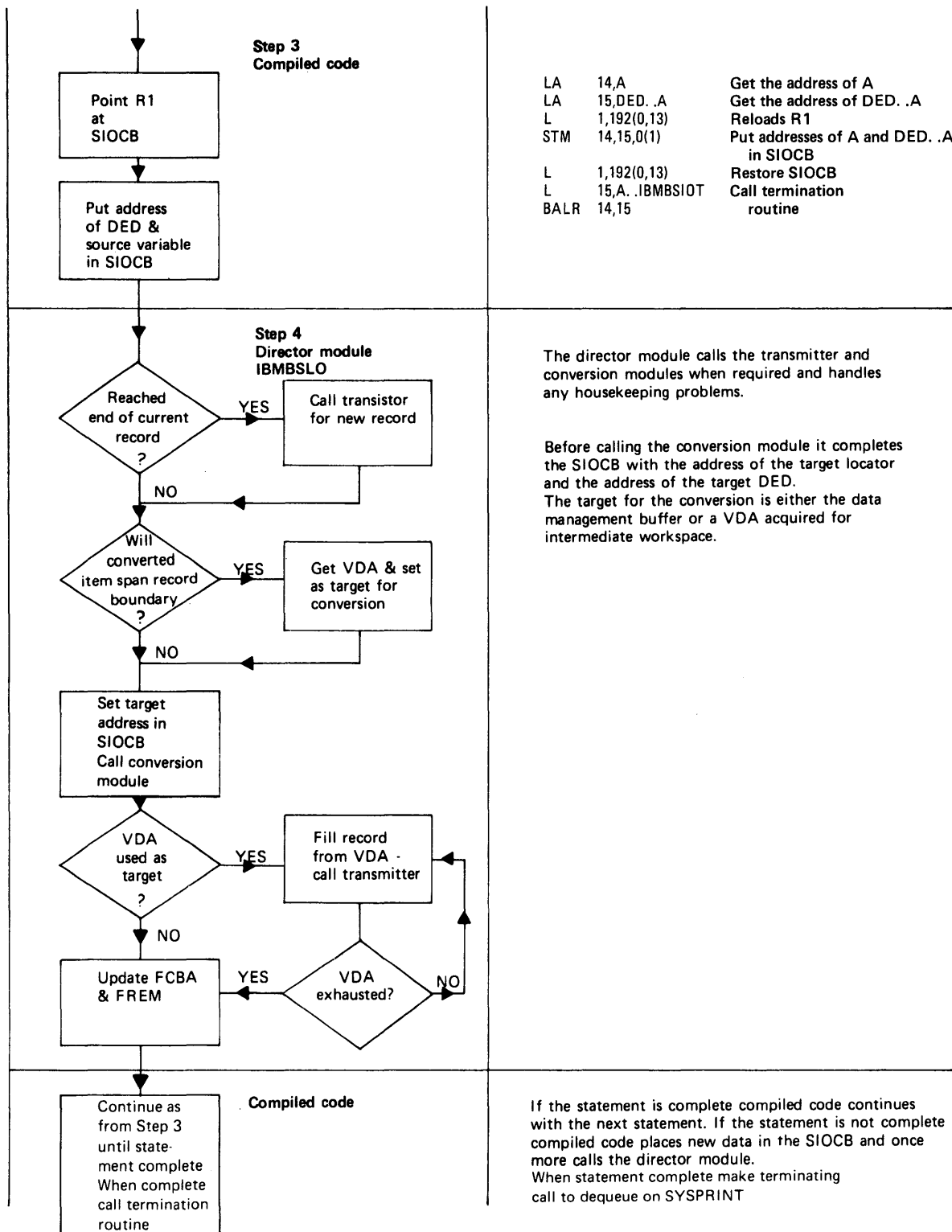
Figure 80 (Part 2 of 2). Flow of Control through a PUT LIST Statement

```
PL/I Source Statements:
        DCL A,B STATIC;
        PUT LIST (A,B);
```

```
* STATEMENT NUMBER   3
00005E  41 70 D 0C8    LA    7,200(0,13)       Pick up address of SIOCB
000062  50 70 3 04C    ST    7,76(0,3)         Store in parameter list
000066  96 80 3 04C    OI    76(3),X'80'       Mark end of parameter list
00006A  41 10 D 0C8    LA    1,200(0,13)       SIOCB pointer
00006E  50 10 D 0C0    ST    1,192(0,13)          to temporary pointer
000072  92 40 D 0D9    MVI   217(13),X'40'     Set LIST OUTPUT flag in SIOCB
000076  41 10 3 048    LA    1,72(0,3)         Point R1 at SIOCB
00007A  58 F0 3 02C    L     15,A..IBMBSIOA    Branch to initializing module
00007E  05 EF          BALR  14,15
000080  41 E0 D 0B8    LA    14,A              Pick up address of A
000084  41 F0 3 038    LA    15,DED..A         Pick up address of DED..A
000088  58 10 D 0C0    L     1,192(0,13)       Restore SIOCB address
00008C  90 EF 1 000    STM   14,15,0(1)        Store addresses in SIOCB
000090  58 F0 3 034    L     15,A..IBMBSLOA    Call list-directed director
000094  05 EF          BALR  14,15                routine
000096  41 E0 3 050    LA    14,B              Pick up address of B
00009A  58 10 D 0C0    L     1,192(0,13)       Point R1 at SIOCB
00009E  50 E0 1 000    ST    14,0(0,1)         Place address B in SIOCB
0000A2  58 F0 3 034    L     15,A..IBMBSLOA    Call list-directed director
0000A6  05 EF          BALR  14,15                routine
0000A8  58 10 D 0C0    L     1,192(0,13)       Point R1 at SIOCB
0000AC  58 F0 3 030    L     15,A..IBMBSIOT    Make terminating call to
0000B0  05 EF          BALR  14,15                dequeue on SYSPRINT
```

            Note: The DEDs for A and B have been
                   commoned. Consequently the same
                   address is kept in the SIOCB for
                   both calls to the director modules.

Figure 81. Code Generated for Typical List-Directed I/O Statement

GET LIST Statement

GET LIST statements follow the same sequence, but the
transmission is in the opposite direction. The main differences
are:

• If record spanning is involved, the item is assembled in
  intermediate workspace before it is converted.

• A locator is built for the source string from the input, and
  the addresses of the locator and of a character DED for the
  source are placed in the SIOCB by the director module. The
  address of the target or its locator and the address of the
  target DED are placed in the SIOCB by compiled code.

• Unless the COPY option is being used, no final call is made
  to the initialization/termination routine.

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 all the variables involved in the statement rather than one variable at a time, and handles the complete statement without returning to compiled code.

The data-directed director module handles the reading or writing of the names, the equals signs, and the punctuation, and then calls the list-directed director module to handle the value for each variable.

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, which 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. For output, the director module completes the statement with a final semicolon. Figure 82 on page 199 shows the complete process.

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.

GET DATA (A,B);

FLOW DIAGRAM

COMPILED CODE & NOTES

Step 1
Compiled code

```
LA    7,216(0,13)       Pick up address of SIOCB
ST    7,84(0,3)         Place in parameter list
OI    84(3),X'80'       Flag last argument in parameter list
LR    1,7               Get address of parameter list
MVI   233(13),X'84'     Set flag DATA INPUT in SIOCB
MVI   234(13),X'01'     Reset flag value
LA    14,104(0,3)       Set abnormal return
ST    14,240(0,13)      Store address in SIOCB
LA    1,80(0,3)         Point R1 at parameter list
ST    15,A..IBMBSIIA    Branch to stream
BALR  14,15               initializing module
```

Set up parameter
list, call
initializer

Step 2
Input initializing module
IBMBSII

Set fields
in SIOCB

The input initializing module is passed the
address of the SIOCB and the FCB for the file.

File
open?   NO   Call IBMBOCL
                 to open file

It checks the validity of the statement, opens
the file and places the address of the FCB in the
SIOCB and returns to compiled code

YES

Return to
compiled code

Step 3
Compiled code

Set up p. list
for data
director
consisting of
A(SIOCB)
A(SYMTAB,I)
A(SYMTAB,J)

```
LA    15,216(0,13)      Pick up address of SIOCB
ST    15,88(0,3)        Place address in parameter
                        list

                        The parameter list contains
                        addresses of symbol tables
                        and variables already set up
                        in static.

LA    1,88(0,3)         Point R1 at parameter list
L     15,A..IBMBSDIA    Call data-directed director
BALR  14,15               module
CL.2 EQU *              Abnormal locate return
                        address
```

Figure 82 (Part 1 of 2). Handling a GET DATA Statement

```
From Step 6
    ┌──┐
    │ A│
    └──┘
                            Step 4
                            Data - directed director
                            module IBMBSDI

        ╱ New ╲      YES    ┌──────────────┐
       ╱ record ╲──────────→│ Call transmitter│
       ╲or spanning?╱       │ setting up VDA │
        ╲        ╱          │ if necessary   │
           │                └──────────────┘
          NO

        ╱ Name ╲
       ╱ in data ╲   NO     ┌──────────────┐
      ╱ stream match╲──────→│ Call IBMBERR │
      ╲  SYMTAB? ╱          └──────────────┘
        ╲      ╱
           │
          YES
    ┌──────────────┐
    │ Place address │
    │ DED and variable│
    │ in SIOCB     │
    └──────────────┘

    ┌──────────────┐
    │ Update FREM & │
    │ FCBA to beyond│
    │ equal symbol │
    └──────────────┘

    ┌──────────────┐
    │ Call list-directed│
    │ director module│
    └──────────────┘
                            Step 5
                            List directed director
    ┌──────────────┐        module IBMBSLI
    │ Decide on    │
    │ conversion   │
    │ required and call│
    │ correct module│
    └──────────────┘

    ┌──────────────┐
    │ Update FREM & │
    │ FCBA         │
    │ Return to    │
    │ IBMBSDI      │
    └──────────────┘
                            Step 6
                            Return to IBMBSDI
    ┌──────────────┐
   ╱│ Repeat from  │
  ╱ │ step 4 until │
 │ A│ final semicolon│
  ╲ │ found        │
   ╲│              │
    └──────────────┘

  Return to compiled code
```

The data directed director module is passed the address of the SIOCB and either a list of symbol table addresses or an address in the symbol table vector.

The module reads in the name, checks that the name read is in the symbol tables passed and if not raises the NAME condition.

When the variable is identified the module places the address of the target and its DED in the SIOCB and calls the list-directed director module passing it the SIOCB.

The list directed module completes the operation as for list directed I/O

On return to the data directed module a search is made for the next name and the action continued as from step 4 until a semicolon is reached in the input stream
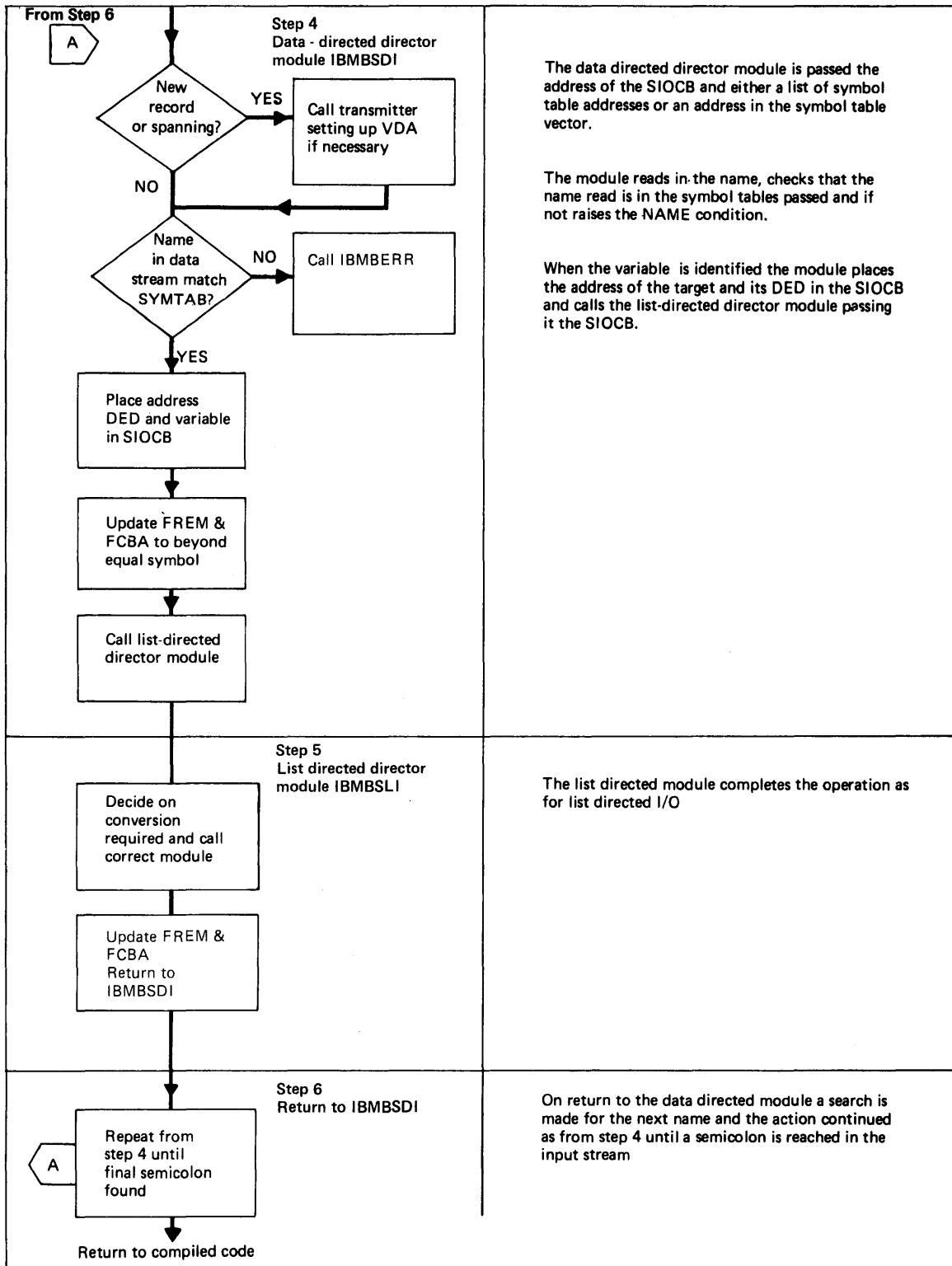
Figure 82 (Part 2 of 2).   Handling a GET DATA Statement

Identifying the Name

If a data list is included in the statement, for example:

        PUT DATA (A,B,C);

the source or target variables are identified from a list of symbol tables. If a data list is not included in the statement, for example:

        PUT DATA;

the source or target variables are identified from the symbol table vector.

A symbol table associates a name with the address of a variable. The symbol table vector is a list of the symbol tables known in the external procedure. The items in a symbol table vector are 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 A.

The object code produced for a PUT DATA statement is shown in Figure 83 on page 202.

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 (for example, F(3), A(25), etc.), known as data format information.

2.   Spacing information (for example, X(3), COL(70), etc.), known as control format information.

Both types of information are compiled as format DEDs (or FEDs) and are passed by compiled code to the routines that require the information.

Because the information is available during compilation, it is possible for the compiler to determine the conversions that will be required. It is consequently possible for compiled code to call the required conversion or conversion director routine, or to generate in-line conversion code without the assistance of a library director module.

```
PL/I source statements:
    DCL A,B,C;
    PUT DATA (A,B,C);
```

RELEVANT SECTION OF THE STATIC INTERNAL STORAGE MAP

```
000048   00000000                A..DCLCB        ┐  Parameter list
00004C   80000000                A..TEMP         ┘  for IBMBSIOA

000050   00000000                A..TEMP         ┐
000054   00000060                A..SYMTAB       │  Parameter list
000058   00000074                A..SYMTAB       │  for IBMBSDOA
00005C   80000088                A..SYMTAB       ┘
000060   8500000100000038        SYMBOL TABLE..A
         000000B800000000
         0001C100
000074   8500000100000038        SYMBOL TABLE..B
         000000BC00000000
         0001C200
000088   8500000100000038        SYMBOL TABLE..C
         000000C000000000
         0001C300
00009C
```

RELEVANT SECTION OF THE OBJECT PROGRAM LISTING

```
* STATEMENT NUMBER   3
00005E   41 70 D 0E8      LA    7,232(0,13)       Pick up address of SIOCB
000062   50 70 3 04C      ST    7,76(0,3)         Store in parameter list
000066   96 80 3 04C      OI    76(3),X'80'       Mark end of parameter list
00006A   18 17            LR    1,7               Place SIOCB in R1
00006C   50 10 D 0E0      ST    1,224(0,13)       Save SIOCB
000070   92 80 D 0F9      MVI   249(13),X'80'     Set data output
000074   92 01 D 0FA      MVI   250(13),X'01'       flags
000078   41 10 3 048      LA    1,72(0,3)         Point R1 at parameter list
00007C   58 F0 3 02C      L     15,A..IBMBSIOA    Call initializing
000080   05 EF            BALR  14,15               routine
000082   41 F0 D 0E8      LA    15,232(0,13)      Pick up address of SIOCB
000086   50 F0 3 050      ST    15,80(0,3)        Place in parameter list
00008A   96 80 D 0FB      OI    251(13),X'80'     Mark end of parameter list
00008E   41 10 3 050      LA    1,80(0,3)         Point R1 at parameter list
000092   58 F0 3 028      L     15,A..IBMBSDOA    Call director routine
000096   05 EF            BALR  14,15
000098   58 10 D 0E0      L     1,224(0,13)       Get SIOCB
00009C   58 F0 3 030      L     15,A..IBMBSIOT    Make terminating call to
0000A0   05 EF            BALR  14,15               dequeue on SYSPRINT
```

Figure 83. Typical Data-Directed Code

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:

- Keeping track of the buffer position, freeing and acquiring intermediate workspace where necessary, and calling the library when a new record is required.

- 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:

- Two short modules, IBMBSEO and IBMBSEI, that interface with the transmitter and are called by the compiler-generated subroutines when a new record is required.

- A routine, IBMBSEDA, that handles the complete processing of an item (as the director does for list-directed I/O). This routine is called when the item cannot be handled by the compiler-generated subroutines.

The decision on whether to use compiler-generated subroutines or the overall library director module is made at compile time. Figure 84 shows the conditions under which each method is used.

```
+--------------------------------+
| Handle entirely by library     |
| routine (IBMBSED), or use       |
| compiler-generated sub-         |
| routines?                       |
+--------------------------------+
```

COMPILER V   LIBRARY   V

```
+-------------------------------+    +------------------------------------------+
| Compiler-generated subroutines |    | IBMBSED handles processing completely for: |
| are used except in the cases   |    |     A-format item with implied length*     |
| shown opposite.  Even so, a     |    |     B-format item with implied length      |
| library routine will be called  |    |     C-format item                          |
| if a new record is required,    |    | *An exception is that A-format items with  |
| and, generally, to handle a     |    | implied length are handled in-line if:     |
| conversion.                     |    | OPT(TIME) is in effect, and the complier   |
|                                 |    | can match the data list with the format    |
+-------------------------------+    | list, and the data item is a character     |
                                      | string.                                    |
                                      +------------------------------------------+
```

Figure 84.   The Use of the Library in Edit-Directed I/O

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.

2. A call to a compiler-generated subroutine to check whether a new record is required, and, if so, to call the module IBMBSEI or IBMBSEO to acquire a new 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.

4. A further call to a compiler-generated subroutine, to update the buffer control fields, and free any intermediate workspace if spanning was involved.

5. A terminating call to the initialization/termination routine.

This sequence is illustrated in the annotated flowchart in Figure 85 on page 205. Figure 86 on page 207 shows the code generated for a GET EDIT statement.

PUT EDIT (B)(A);

FLOW DIAGRAM

Set up part of SIOCB. Call initialization routine IBMBSIO	**Step 1** Compiled code

Step 2 Initialization routine

File closed — YES → **Call IBMBOCL to open file & call transmitter to get 1st record**

NO

Check statement validity

Place address of variable, its DED, & DED generated from format item, in SIOCB	**Step 3** Compiled code

X

Call IELCGOA

Step 4 IELCGOA

Will item span or require new record? — YES → **Set 'VDA' flag in SIOCB. Get VDA and set as address of target.**

A

NOTES

LA	7,216(0,13)	Pick up address of SIOCB
ST	7,84(0,3)	Place in parameter list
OI	84(3),X'80'	Mark end of parameter list
LR	1,7	Point R1 at SIOCB
ST	1,208(0,13)	Save in temp
MVI	233(13),X'20'	Set EDIT OUTPUT flag
LA	1,80(0,3)	Point R1 at parameter list
L	15,A..IBMBSIOA	Branch to initialization
BALR	14,15	routine

Test if file is open, and open if necessary, calling transmitter to locate record.
Place address of ONCA and FCB in the SIOCB.
Check statement validity.

LA	14,B	Get address of data
LA	15,DED..B	Get address of DED..B
L	1,208(0,13)	Get SIOCD address
STM	14,15,0(1)	Place addresses of B and DED..B in SIOCB
LA	14,68(0,3)	Get address of FED
L	7,A..IELCGOG	Branch to compiler-generated
BALR	6,7	subroutine

Acquire VDA for item if necessary.
Either if there is no room in current record, or,
if the converted item will span the record boundary.

Figure 85 (Part 1 of 2). Edit-Directed Statement with Matching Data and Format Lists

A

FLOW DIAGRAM   NOTES

Step 5
Carry out   **Compiled code**   L   15,A. .IBMBSAOA   Call output conversion
conversion either   **or conversion**   BALR   14,15   director (A-format)
in - line or by   **routine**
calling library module   L   7,A. .IELCGOH   Call compiler-generated
                                                               BALR   6,7   director routine
Call
IELCGOB   L   1,208(0,13)   Get SIOCB address

Step 6
IELCGOB   Update buffer control fields
Item   Handle housekeeping
handled by   YES
IBMBSEDB
?

NO

Was a   YES   Call IBMBSEOA
VDA used?   Call transmitter
                            and free VDA

NO

Update FREM,   Clear 'VDA' flag
FCBA, and FCNT   and IBMBSED
                           flag

Return to
compiled code

Step 7
Continue from   **Compiled code**   Continue as necessary
STEP 3 with next   When complete call termination
item, if any   routine to dequeue on SYSPRINT
When complete
make terminating   L   15,A .. IBMBSIOT
call to IBMBSIOT   BALR   14,15

Figure 85 (Part 2 of 2).   Edit-Directed Statement with Matching Data and Format
                            Lists

```
PL/I source statements:
      DCL A,B;
      GET EDIT (A,B) (F(3), X(8));
```

```
* STATEMENT NUMBER  3
00005E  41 70 D 0D8    LA    7,216(0,13)       Pick up address of SIOCB
000062  50 70 3 05C    ST    7,92(0,3)         Store in parameter list
000066  96 80 3 05C    OI    92(3),X'80'       Mark end of parameter list
00006A  18 17          LR    1,7               Place SIOCB in R1
00006C  50 10 D 0D0    ST    1,208(0,13)       Save SIOCB
000070  92 24 D 0E9    MVI   233(13),X'24'     Set EDIT INPUT flags in SIOCB
000074  41 E0 3 060    LA    14,96(0,3)        Pick up return address (CL.2)
000078  50 E0 D 0F0    ST    14,240(0,13)      Store in SIOCB
00007C  41 10 3 058    LA    1,88(0,3)         Point R1 at parameter list
000080  58 F0 3 038    L     15,A..IBMBSIIA    Call stream I/O
000084  05 EF          BALR  14,15               initialization routine
000086  41 E0 D 0B8    LA    14,A              Pick up address of data
00008A  41 F0 3 040    LA    15,DED..A         Pick up address of DED..A
00008E  58 10 D 0D0    L     1,208(0,13)       Get SIOCB address
000092  90 EF 1 008    STM   14,15,8(1)        Puts addresses of A and DED..A
                                                 in SIOCB
000096  41 E0 3 044    LA    14,68(0,3)        Point R14 at FED
00009A  58 70 3 014    L     7,A..IELCGIX      Call compiler-generated
00009E  05 67          BALR  6,7                 subroutine
0000A0  58 F0 3 034    L     15,A..IBMBSFIA    Call conversion director routine
0000A4  05 EF          BALR  14,15
0000A6  58 70 3 018    L     7,A..IELCGIB      Call compiler-generated
0000AA  05 67          BALR  6,7                 subroutine
0000AC  41 E0 3 04A    LA    14,74(0,3)        Pick up FED of X format item
0000B0  58 10 D 0D0    L     1,208(0,13)       Pick up address of SIOCB
0000B4  58 70 3 014    L     7,A..IELCGIX      Call compiler-generated
0000B8  05 67          BALR  6,7                 subroutine
0000BA  41 E0 D 0BC    LA    14,B              Pick up address of B
0000BE  50 E0 1 008    ST    14,8(0,1)         Store in SIOCB
0000C2  41 E0 3 044    LA    14,68(0,3)        Point R14 at FED
0000C6  58 70 3 014    L     7,A..IELCGIX      Call compiler-generated
0000CA  05 67          BALR  6,7                 subroutine
0000CC  58 F0 3 034    L     15,A..IBMBSFIA    Call conversion director routine
0000D0  05 EF          BALR  14,15
0000D2  58 70 3 018    L     7,A..IELCGIB      Call compiler-generated
0000D6  05 67          BALR  6,7                 subroutine
0000D8          CL.2   EQU   *                 Abnormal return address
```

Figure 86.   Code Generated for an Edit-Directed Statement with Matching Data and
              Format Lists

Handling Control Format Items

Control format items are implemented by calling a formatting module, and passing it the SIOCB containing the address of an FED for a control format item. There are four formatting modules:

IBMBSPL   Library routine for SKIP, PAGE, and LINE formats and options.

IBMBSXC   Library routine for X and COLUMN formats.

IELCGOC   Compiler-generated subroutine for X output items that do not span a record boundary.

IELCGIX   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 Nonmatching 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 specific conversions for specific variables. However, it is possible to write statements which, because 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 possible 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 code for data items and the code for format items as the value 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 sequence used for matching and non-matching data and format lists are shown in Figure 87 on page 209.

MATCHING LISTS

PUT EDIT (I, NAME, ACT. NO)
(F (3),X (3), A (15), X (3), P'ZZZ9');

```
┌─────────────┐
│ HANDLE      │
│ CONVERSION  │
│ OF I        │
└─────────────┘
       │
       ▼
┌─────────────┐
│ HANDLE      │
│ X ITEM      │
└─────────────┘
       │
       ▼
┌─────────────┐
│ HANDLE      │
│ CONVERSION  │
│ OF NAME     │
└─────────────┘
       │
       ▼
┌─────────────┐
│ HANDLE      │
│ X ITEM      │
└─────────────┘
       │
       ▼
┌─────────────┐
│ HANDLE      │
│ CONVERSION  │
│ OF          │
│ ACT · NO    │
└─────────────┘
       │
       ▼
```
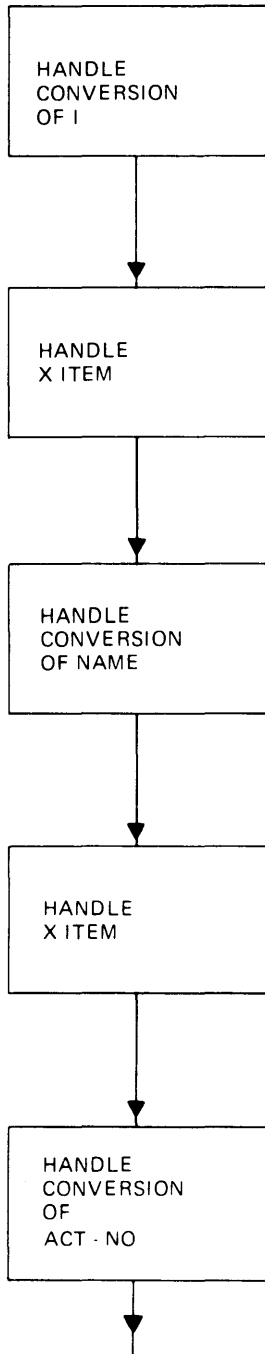
UNMATCHING LISTS

PUT EDIT (AB, C, D) ((N) F (3), SKIP, A (4));
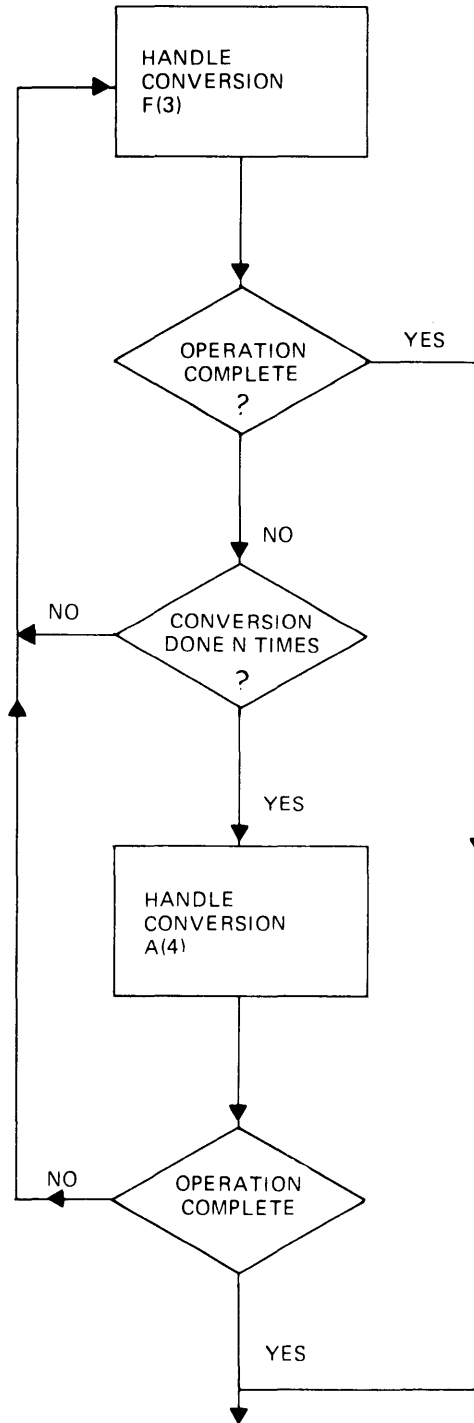
Figure 87. Code Sequences Used for Matching and Nonmatching Data and Format Lists

FORMATTING FOR PRINT FILES

Formatting information such as page size, line size, page length and tab positions for print files are accessed by list and data-directed director modules from a field TTAB held at offset X'50' in the TCA. The field holds the address of the tab table to be used. That is, either the PLITABS control section, if provided by the user, or the IBMBSTAB control section, if the default is to be used.

The control section PLITABS can be provided by the user either as a control section which is link-edited with the object module or as a PL/I structure declared in his program as PLITABS. This structure is then compiled as a suitable control section by the optimizing compiler.

The programmer may also use the default which is provided as a transient library module loaded by the open routines. The format of PLITABS and its default values are given in the programmer's guide for this compiler.

When the open routines are called, they inspect the TCA to determine whether PLITABS has been provided by the user. If it has not, they load the transient library routine IBMBSTAB, which holds the default tab setting. When the routine is loaded, the address of entry point IBMBSTAB is placed in the TTAB field in the TCA. If PLITABS has been provided by the user, its address will have been placed in TTAB by the linkage editor.

HANDLING FORMAT OPTIONS

Format options (for example, GET SKIP(4), PUT PAGE, GET SKIP LIST) are handled by a call to the appropriate entry point of the initialization routine.

The initializing module calls the formatting module IBMBSPL to carry out the formatting.

INPUT AND OUTPUT OF COMPLETE ARRAYS

When transmitting complete arrays, it is not economical 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, the necessary indexing being carried out by compiled code.

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 input 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.

TRANSMIT Condition

The rules for raising the TRANSMIT condition in stream I/O are that the condition shall be raised after the assignment or output 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 during the processing of the statement involved.

When the TRANSMIT condition is detected by the data management routines, control is passed to the error routine in the transmitter, which sets a flag in the FCB indicating a transmission error. For input, the director module inspects

this flag, and, if it is set, sets a flag in the SIOCB.
TRANSMIT is raised for every item that is taken from a record in
the block with which the transmission error was associated. It
is raised after each potentially incorrect value has been
assigned. For output, TRANSMIT is raised by the transmitter as
soon as 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, IBMBSEIT 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, IBMBSCVA. This module analyzes the type
of conversion error, and calls the error handler with an
appropriate error code. For input, 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 a
further GET statement was executed on the same file 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, or when the name is
unobtainable (it might, for example, be out of subscript range.)
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 transmitter routines, which then
enter the SYNAD routine in the transmitter. This routine sets a
flag in the FCB. On return to the director modules, the flag is
tested and, depending on the situation in which the transmitter
was called, ENDFILE or unexpected end of file is raised by
calling the error handler.

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 for the statement is kept in the
SIOCB, and then copied into the FCB after every transmission to
or from a PL/I variable.

The DATAFIELD built-in function is handled by the data-directed
director routine, which places the address of the string
locator/descriptor for the offending field in the ONCA. The
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.

THE COPY OPTION

The COPY option allows input data to be copied onto a specified output file. At the start of a GET statement with the COPY option, a flag is set in the FCB, and the current buffer position is saved in the field FCPM in the FCB.

A resident library routine, IBMBSCP, is used to handle the data, and to transmit it to the copy file by calling the appropriate transmitter. IBMBSCP is called at the end of the GET statement, and during the statement if a new buffer is acquired. As shown in Figure 88, the data transmitted to the copy file is that which is held between the pointers FCPM and FCBA. FCBA points to the next byte to be read; FCPM points to the start of the data to be copied. FCPM is updated to point to the start of the new buffer when a transmitter call is made during the execution of the statement. The copy flag is turned off during the terminating call to IBMBSII.

If an interrupt occurs during the execution of a GET statement with the COPY option, it is possible that the terminating call to IBMBSII will be bypassed because of a GOTO from an ON-unit, or because the job is terminated. For this reason, a test is made on the copy flag at the start of every GET statement, and when the file is closed. If the copy flag is on, IBMBSCP is called to handle the data. When the data has been transmitted, the flag is turned off.

Handling the Copy File

During the initializing call, IBMBSII determines whether the copy file is open and, if it is not, calls IBMBOCL to open the file. The address of the DCLCB for the copy file is then stored in the FCB of the input file. The data is transmitted to the file by calling the transmitter for the file type.

GET LIST FILE (SYSIN) (STRING1)
COPY FILE(A):

GET LIST FILE (SYSIN) (STRING2)
COPY FILE (A);

GET LIST FILE (SYSIN) (STRING3)
COPY FILE (B);

FCPM — pointer for start of COPY data

'DATA FOR COPYING ONTO'   'FILE   NAMED A'   'DATA FOR COPYING ONTO FILE B'

pointer for end of COPY data

FCBA

Data is transmitted to the copy file at the end of each statement and at those
times when it can no longer be held between the pointers FCBA and FCPM.
In the example above this will be at the end of each GET statement and at
the end of the first record.

Figure 88. The Current Buffer Pointer FCBA and FCPM, the Copy Pointer, Keep Track
of the Data to be Copied

THE STRING OPTION

The STRING option allows data to be transmitted between a string and one or more PL/I variables by means of a stream I/O statement.

The STRING option is implemented by treating the string specified in the statement as if it were the buffer, and the other PL/I variables as if they were the sources or targets. The difference in housekeeping between string and file operations is resolved by the use of a string housekeeping routine, IBMBSIS. IBMBSIS is called in the place of the stream I/O initialization/termination routine. IBMBSIS sets up a dummy FCB that is initialized so that the correct action is taken should the director modules attempt to read or write beyond the end of the string. After the dummy FCB has been initialized, the director modules are called to convert and move the data as in normal stream I/O.

To implement the string option, compiled code passes the string housekeeping module an extended SIOCB in which the dummy FCB is created. The buffer control fields FCBA and FREM in the dummy FCB are set up as if the string were a record. The field that, in a normal FCB, would hold the address of the transmitter, holds addresses of other sections of code.

For a PUT STRING statement, the transmitter address field is initialized to point to the error handler. Register 1 will have been pointed to the head of the FCB by the caller. The error code for exceeding string size is, therefore, placed at the head of the FCB, and the correct error condition is automatically raised when the branch to the error handler is made.

For a GET STRING statement, the address in the transmitter field is the address of code that sets the end-of-file flag and returns to the caller. This code is held within the dummy FCB.

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 a stream I/O statement. Where the ENDFILE condition or unexpected end of file would be raised for a stream file, a 'GET STRING SIZE EXCEEDED' message is generated, and the ERROR condition is raised.

Completing String-Handling Operations

One or more further calls may be made to the string housekeeping routine IBMBSIS at entry point T, to update the string characteristics after a data item has been transmitted.

PUT STATEMENTS WITH FIXED-LENGTH STRINGS: IBMBSIS is called after the first item has been assigned to the string, to pad the remainder of the string with blanks.

PUT STATEMENTS WITH VARYING STRINGS: IBMBSIS is called to update the length of the string after each item is transmitted.

GET STATEMENTS WITH VARYING STRING: IBMBSIS is always called.

The need to make a further call to IBMBSIS is flagged in the SIOCB when IBMBSISA is called to initialized a statement. The library director routines and the compiler-generated subroutines test this flag, and call IBMBSIS if necessary.

THE CONVERSATIONAL SYSTEM AND CONVERSATION FILES

When using a conversational system, the PL/I programmer can attach his terminal as the input or output device used by one or more stream files.

Three transient library routines are used to implement this facility. Two are transmitters that are used to interface with the conversational system using the appropriate macro instructions, or simulations of them for CMS, to effect the input and output. They also poll for attention interrupts. The third module is a formatting module that overcomes the special formatting difficulties that arise when working at a terminal.

When the file is opened, the OPEN routine tests every stream I/O file to see whether it is to be associated with a terminal. If the file is to be associated with a terminal, the appropriate conversational transmitter loaded:

    IBMBSIC for input
    IBMBSOC for output

A flag is set in the FCB of the file to indicate that the file is a conversational file

The two transmitter modules handle the input, output, and prompting. Formatting differences between conversational and normal I/O are handled by a transient library routine, IBMBSPC. This routine is called by the formatting routine, IBMBSPL, when a conversational file is being handled.

If a conversational module is used, its address is placed in the TCA loaded-module list.

CONVERSATIONAL TRANSMITTER MODULES

Output Transmitter IBMBSOC

The output module IBMBSOC is similar to other output transmitters except that it interfaces with TSO, and uses the TPUT macro instruction. For CMS it uses a simulation of TPUT. The macro instruction is used with the WAIT option to ensure proper queueing of output to the terminal.

Input Transmitter IBMBSIC

The input transmitter carries out a similar function to other PL/I input transmitters. However, it also has to handle certain prompting functions, and implements certain facilities required only for conversational output.

INPUT: Input is achieved by issuing a TGET macro instruction to the TSO control program. For CMS it uses a TGET simulation.

PROMPTING: Prompting is carried out before every input statement, unless the last character transmitted to the foreground terminal was a colon. At the start of a statement, the prompting sequence is: skip to a new line, print a colon, and skip to the start of the next line. If the GET statement is not completed by the data transmitted from the terminal, a further call to the transmitter will be made by the director module handling the stream I/O. A further prompt is then issued to the programmer. Second and subsequent prompts take the form of a plus character followed by a colon.

Prompts are issued by placing the required prompt-code in a suitable field, and using a TPUT macro instruction with a HOLD option. This ensures that any terminal output from previously executed PUT statements will appear at the terminal before the user is prompted to enter his input.

The prompt is issued to the foreground terminal irrespective of whether a PL/I output file is associated with the terminal.

To simplify terminal usage various methods of data input are allowed that do not conform strictly to PL/I language specifications. For example list-directed input need not have a delimiting comma or blank and the trailing blanks need not be entered if a character item in edit-directed I/O does not fill the specified field width.

Formatting Module IBMBSPC

To simplify the use of a terminal, default formatting conventions are assumed. These apply to PAGE, SKIP, and LINE instructions and can be summarized as follows:

- SKIP instructions of 3 lines or less are followed.

- PAGE and LINE and SKIP instructions of more than 3 lines are interpreted as SKIP(3) instructions.

This default formatting can be overridden by the use of a PLITABS structure that specifies a value of 1 or greater for the page length. (PLITABS is described above under the heading "Formatting for Print Files.")

IBMBSPC checks the page-length value in the PLITABS control section. This control section will be either the default taken from the PL/I transient library module IBMBSTAB, or, if the values have been specified by the programmer, will be the values in the structure declared with the name PLITABS, or, possibly, a link-edited control section called PLITABS. In the library module IBMBSTAB, the page-length value is zero.

If the page-length value in the PLITABS control section is zero, the formatting conventions described above are used. These are referred to as squashed mode. If the value is greater than zero, normal formatting is undertaken.

The method of formatting used is for IBMBSPC to insert the required number of 'new line' characters in the output buffer, and to call the transmitter to transmit the buffer contents. (In the special case of SKIP (0), backspace characters are used.

The normal PL/I rules for ENDPAGE apply to formatted terminal output. ENDPAGE is not raised for squashed mode output.

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.

Ten 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. Conversational modules

8. The conversion fix-up module (IBMBSCV)

9. The copy module (IBMBSCP)

10. The string housekeeping module (IBMBSIS)

Conversion modules are described in Chapter 10, "Data
Conversion" on page 220. 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:

IBMBSII   Input initializer

IBMBSIO   Output initializer

A further module is used for string handling, which is listed
under "Miscellaneous Modules" on page 219.

IBMBSII is discussed in "The COPY Option" on page 212, while
IBMBSIO is described under "PUT LIST Statement" on page 193.

DIRECTOR MODULES

Library Director Routines

IBMBSLI   List-directed input

           Entry point A: element item

           Entry point B: complete array

IBMBSLO   List-directed output

           Entry point A: element item

           Entry point B: complete array

IBMBSDI   Data-directed input

           Entry point A: with data list

           Entry point B: all known variables

IBMBSDO   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 output

           Entry point D: CHECK output for a single item

           Entry point T: output a final semicolon only.

Modules Used with Compiler-Generated Subroutines

IBMBSEI   Edit-directed input

> Entry point A: housekeeping for input item
> spanning a record boundary.

> Entry point T: raise TRANSMIT for spanning input
> item

IBMBSEO   Edit-directed output housekeeping for output item
spanning a record boundary.

Module for Complete Library Control of Edit-Directed I/O of a Single Item

IBMBSED

> Entry point A: edit-directed input

> Entry point B: edit-directed output

Compiler-Generated Director Routines

For input:

IELCGIX   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 the VDA if one was used, updating the
COUNT built-in function value, and calling IBMBSEIT if
TRANSMIT has been raised.

For output:

IELCGOG   Provides the address of the target of an edit-directed
data item.

IELCGOH   Completes the transmission of an edit-directed data
item, updating the buffer items in the DCLCB, 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 routines of OS data
management. These modules essentially complete the setting up
of the DCB, 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:

IBMBSTI   Input transmitter

IBMBSOF   Output transmitter for F-format records

IBMBSOV   Output transmitter for V-format records

IBMBSOU   Output transmitter for U-format records

IBMBSTF   Print transmitter for F-format records

IBMBSTV   Print transmitter for V-format records

                    IBMBSTU   Print transmitter for U-format records

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

                    IBMSBPL   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

                    IBMBSXC   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

                    IELCGOCA   X items, in edit-directed output, that do not span a
                               record boundary.

EXTERNAL CONVERSATION 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

CONVERSATIONAL MODULES

          Transmitters:

          IBMBSIC   input transmitter

          IBMBSOC   output transmitter

          Formatting module:

          IBMBSPC   formatting module

MISCELLANEOUS MODULES

The other subroutines used in stream I/O are:

IBMBSCV   the conversion fix-up module

IBMBSCP   the copy module

IBMBSIS   the string housekeeping module

| I/O UNDER CICS

Most input/output operations on CICS are handled by CICS macro or command instructions; however, stream output is supported to the SYSPRINT file by means of transient data on the CPLI queue. Basically, the scheme is similar to normal stream I/O with the director routine in overall control, calling transmitter and conversion modules as they are required. The complete operation is controlled by IBMBSIO which is called at the start and end of the statement.

The major difference from the non-CICS system is that the standard open/close modules and transmitter are not used. Instead a single module IBMFSTV is used for both operations. The operations are logically separate but are held in one module to reduce the number of loads required.

The major difference between CICS and the OS implementation is that for CICS an FCB is set up by IBMFSTV, whereas for OS, the FCB is set up by the library OPEN modules.

Control passes to the CICS module because DFHPL1OI contains dummy entry points which correspond to the standard entry points of IBMBOCL. The dummy entry points pass control to code in DFHSAP that loads IBMFSTV and passes control to the relevant entry point in IBMFSTV. Only one test for CICS needs to be made, and that is before a test is made to see if the file is open. For CICS the open file chain is tested instead of the normal control blocks.

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.

INTRODUCTION

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 optimization has been specified, all conversions that can be handled by in-line code are so handled. If optimization has not been specified, the simpler and more commonly used conversions will be handled in-line, the remainder by the library conversion package.

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 89 on page 221.

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<p=16: long floating-point p>16: extended floating-point
FLOAT BINARY(p)	p<=21: short floating-point p>21<p<=53: long floating-point p>53: extendedfloating-point
PICTURE	One byte for each picture character (except K and V)

Figure 89. Internal Forms of Data Types

THE LIBRARY CONVERSION PACKAGE

The library conversion package consists of some 26 modules and is capable of handling all the conversions that are allowed in the OS 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. 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 OS 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.

Conversion Module Naming Conventions

All names begin with the letters 'IBMB'. The fifth letter is 'C' for conversions, conversion utilities, and the string/arithmetic directors. It is 'S' for the edit-directed format directors. The modules in the arithmetic conversion package have six letter names, the sixth letter being an arbitrary module identifier. The string conversion modules and conversion utilities have seven letter names in which the sixth and seventh letters are mnemonic. The mnemonic codes follow:

X	extended float
F	float
I	integer or binary constant if in C module
I	input if in S module
D	fixed decimal
Z	free decimal or float decimal
P	fixed pictured decimal
E	float pictured decimal
H	decimal constant
Y	float decimal constant on output
B	bit
J	bit constant
C	character
Q	pictured character
A	arithmetic
O	output in S module
G	"check" or utility
T	table

SPECIFYING A CONVERSION PATH

When a number of conversion modules need to be used for a certain conversion, some control of the path taken is necessary 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 arithmetic 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 can use the previous module's workspace. 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 same workspace can be used. 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 expects a standard set of arguments.
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
"Data Element Descriptors (DEDs)" on page 64, and fully mapped
in "Data Element Descriptor (DED)" on page 337.

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 and DEDs 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 using the same workspace, registers are normally used
rather than a parameter list.

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 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.

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 90 on page 224.

Conversion Source	Conversion Target	Comments and Conditions	Optimization SIZE Disabled	Optimization 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 nonzero, 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

Figure 90 (Part 1 of 2). Data Conversions Performed In-line

Conversion Source	Conversion Target	Comments and Conditions	Optimization SIZE Disabled	Optimization SIZE Enabled
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 length ≤ 2048.	time	not done in-line
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 (see note below)	Fixed binary	Source precision must be < 10.	time	not done in-line
	Fixed decimal	If picture has a sign, the optimization must be 'time'.	–	not done in-line
	Floating-point	Source precision must be < 10.	time	not done in-line
	Picture	Picture type 1, 2, or 3.	time	not done in-line
Locator	Locator	–	–	–
Label	Label	–	–	–

Figure 90 (Part 2 of 2).   Data Conversions Performed In-line

Note: 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.

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 91 on page 226.

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

Figure 91.   Fundamental In-line Conversions

Note to Figure 91:   Conversions 1, 11, and 13 are not used.

Note about Picture Variables

Not all the picture characters available may be used in a picture involved in an in-line arithmetic conversion.   The only ones permitted are:

V and 9

Drifting or nondrifting characters $ S + -

Zero suppression characters Z *

Punctuation characters , . / B

For in-line conversions, pictures with this subset of characters are divided into three types:

Picture type 1:
Pictures of all 9s with (optionally) a V and a leading or trailing sign.   For example:

'99V999', '99', 'S99V9',
'99V+', '$999'

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) punctuation characters and a sign character. For example:
>
>   '$$$$', '-,--9', 'S/SS/S9',
>   '+++9V.9', '$$$9-'

Sometimes a picture conversion is not performed in-line even though the picture is one of the above types, because it has certain characteristics that necessitate a subroutine call. These may be, for instance:

* 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
      in temp.

* Punctuation between a drifting Z or a drifting * and the first 9 is not preceded by a V. For example:

      'ZZ.99'

* Drifting or zero suppression characters to the right of the decimal point. For example:

      'ZZV.ZZ', '++V++'

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     14,SOURCE
            LTR   14,14                 Determination
            BNM   Compiler label        Branch if >0
            A     14, Constant          Add a constant to negative
                                          source
Compiler label EQU *
            SRA   14,9                  Divide by source scale
                                          (2**9)
            CVD   14,WKSP+8             Convert to decimal in
                                          workspace
            XC    TARGET(3),TARGET      Set zeros in target
            MVC   TARGET+3(5),WKSP+8    Transfer value to target
            MVN   TARGET+7(1),WKSP+15   Transfer the sign
```

MULTIPLE CONVERSIONS

The conversions listed in Figure 91 on page 226 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 92.

Conversion Required	Number	Compiler Conversions Used
Fixed-decimal to bit-string	7	Fixed-decimal to fixed-binary
	15	Fixed-binary to bit-string
Floating-point to bit-string	3	Floating-point to fixed-binary
	15	Fixed-binary to bit-string
Bit-string to fixed-decimal	17	Bit-string to fixed-binary
	6	Fixed-binary to fixed-decimal
Bit-string to floating-point	17	Bit-string to fixed-binary
	2	Fixed-binary to floating-point
Character-string to bit-string	10	Character-string to fixed-binary
	15	Fixed-binary to bit-string
Fixed-binary to character-string	6	Fixed-binary to fixed-decimal
	12	Fixed-decimal to character-string
Fixed-binary to decimal picture	6	Fixed-binary to fixed-decimal
	18, 19, or 20	Fixed-decimal to picture
Floating-point to decimal picture	5	Floating-point to fixed-decimal
	18, 19, or 20	Fixed-decimal to picture
Decimal picture to fixed-binary	21	Picture to fixed-decimal
	7	Fixed-decimal to fixed-binary
Decimal picture to floating-point	21	Picture to fixed-decimal
	4	Fixed-decimal to floating-point
Decimal picture to decimal picture	21	Picture to fixed-decimal
	18, 19, or 20	Fixed-decimal to picture

Figure 92. Multiple Conversions

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 CONVERSION 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 pseudo-variable is used to change the invalid field or character.

Raising the CONVERSION condition 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, IBMBERR, and looks after the housekeeping problems.

IBMBSCV saves considerable overheads 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 IBMBERR.

CHAPTER 11. MISCELLANEOUS LIBRARY SUBROUTINES AND SYSTEM INTERFACES

In addition to employing the PL/I libraries for the functions described in previous chapters, the OS 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 System/360 Operating System: PL/I Resident Library Program Logic and IBM System/360 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.

COMPUTATION AND DATA-HANDLING SUBROUTINES

The computational and data-handling subroutines are used to handle all the mathematical built-in functions, majority of arithmetic built-in functions, and a number of array-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 impractical. Where possible, arithmetic functions are handled by in-line code. The circumstances in which library subroutines are used for arithmetic functions are listed in Figure 93.

Function	Data Type	Module Name	When Used
Real Arguments			
Integer exponentiation	Short floating-point Long floating-point Extended floating-point	IBMBMXS IBMBMXL IBMBMXE	When exponent is a variable When exponent is a variable Always
General exponentiation	Short floating-point Long floating-point Extended floating-point	IBMBMYS IBMBMYL IBMBMYE	Always Always Always

Figure 93 (Part 1 of 2). Arithmetic Operations Performed by Library Subroutines

Function	Data Type	Module Name	When Used
Complex **Arguments** Integer exponentiation	Short floating-point Long floating-point Extended floating-point	IBMBMXW IBMBMXY IBMBMXZ	When exponent is a variable When exponent is a variable Always
General exponentiation	Short floating-point Long floating-point Extended floating-point	IBMBMYX IBMBMYY IBMBMYZ	Always Always Always

Figure 93 (Part 2 of 2). Arithmetic Operations Performed by Library Subroutines

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. Compiled code always passes arguments in a parameter list. 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 address of DEDs). 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 OS 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 subroutines.

The subroutines involved in these functions are shown in Figure 94 on page 232.

Subroutine	Meaning
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)
IBMBAPE	PROD built-in function (extended floating-point)
IBMBAPF	PROD built-in function (short or long floating-point)
IBMBAPM	STRING pseudo-variable
IBMBASC	SUM built-in function (fixed-point)
IBMFASE	SUM built-in function (extended floating-point)
IBMBASF	SUM built-in function (short or long floating-point)
IBMBAYE	POLY built-in function (extended floating-point)
IBMBAYF	POLY built-in function (short or long floating-point)
IBMBBBA	AND and OR logical operations (aligned bit strings)
IBMBBBC	Compare aligned bit strings
IBMBBBN	Invert aligned bit string (NOT)
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) Compare aligned bit strings

Figure 94.   Array, Structure, and String Subroutines

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.

Handling Interleaved Arrays (IBMBAIH)

IBMBAIH is used to assist the other library array-handling subroutines to process 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:

```
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.

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 number of dimensions in the array, the address of the array descriptor, and the address of the work area in which to construct a table. 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 noncontiguous (see Figure 95).

Declaration   Storage

```
DECLARE    1X(2),          ┌─────────────┐
             2 C,          │ C           │
             2 Y (2),      ├─────────────┤
               3 Z (3)     │ Z (1,1,1)   │ M[3]    A          A
               3 B;        ├─────────────┤         │
                           │ Z (1,1,2)   │        M[2]
                           ├─────────────┤         │
                           │ Z (1,1,3)   │         V
                           ├─────────────┤
               Inc[2]      │ B           │        ─────
                           ├─────────────┤
                           │ Z (1,2,1)   │                  M[1]
                           ├─────────────┤
                           │ Z (1,2,2)   │
                           ├─────────────┤
                           │ Z (1,2,3)   │
                           ├─────────────┤
                           │ B           │
                           ├─────────────┤
               Inc[1]      │ C           │                   V
                           ├─────────────┤
                           │ Z (2,1,1)   │
                           ├─────────────┤
                           │ Z (2,1,2)   │
                           ├─────────────┤
                           │ Z (2,1,3)   │
                           ├─────────────┤
               Inc[2]      │ B           │
                           ├─────────────┤
                           │ Z (2,2,1)   │
                           ├─────────────┤
                           │ Z (2,2,2)   │
                           ├─────────────┤
                           │ Z (2,2,3)   │
                           ├─────────────┤
                           │ B           │
                           └─────────────┘
```

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:

$$Inc[i] = M[i] - E[i+1] \times M[i+1] + Inc[i+1]$$
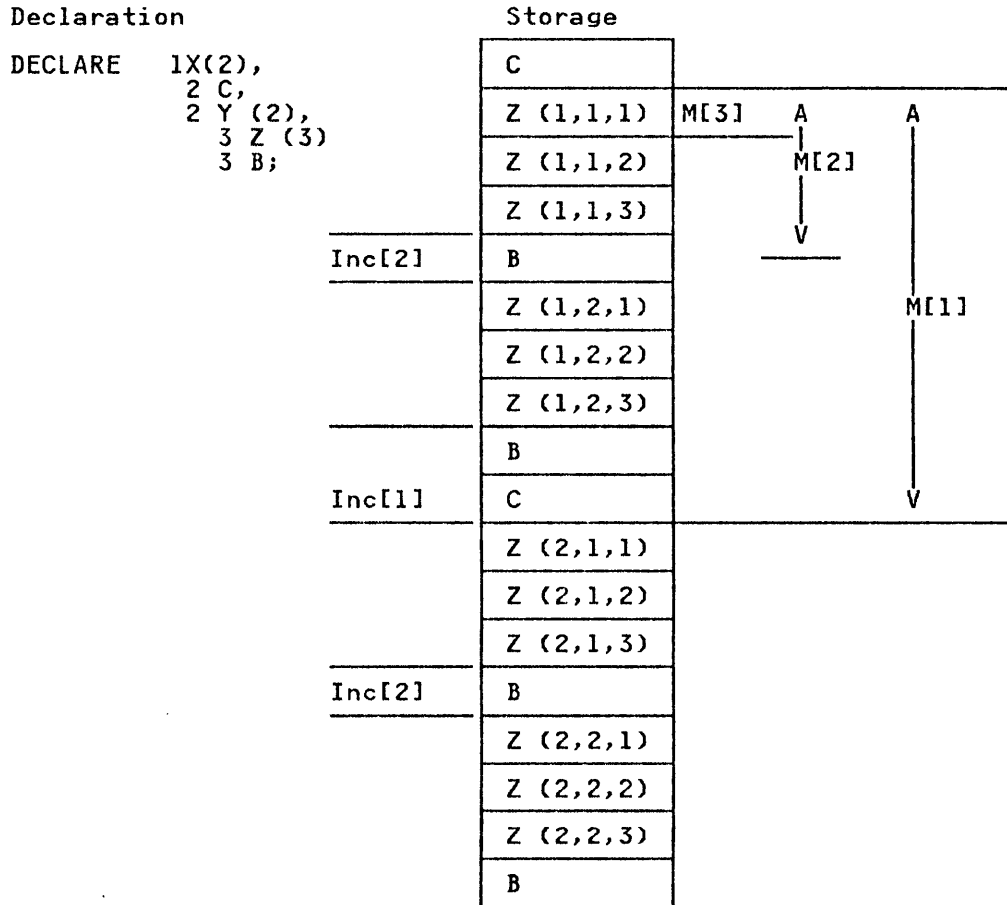
Where $E[i+1]$ is the extent of the (i+1)th dimension.

Figure 95 (Part 1 of 2). Indexing Interleaved Arrays

Increment table for array Z (as initialized by IBMBAIH)

```
                     ┌──────────┐
         ┌──────┐    │ 2        │   subscript count
2nd dimension──> │   ├──────────┤
         │      │    │ 2        │   extent of dimension
         │      │    ├──────────┤
         └──────┼────│ Inc[2]   │   increment
         ┌──────┤    ├──────────┤
         │      │    │ 2        │   subscript count
1st dimension──> │   ├──────────┤
         │      │    │ 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 95 (Part 2 of 2). Indexing Interleaved Arrays

The information in the completed table is used by the 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 communication 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 sort/merge and checkpoint/restart built-in subroutines.

Calls to these facilities are made through library subroutines held in the OS 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.

The OS macro instructions referred to below are described in OS/VS2 MVS Data Management Macro Instructions, or in MVS/Extended Architecture Data Management Macro Instructions.

TIME

The PL/I TIME built-in function is implemented by issuing a
GETIME macro instruction. This is done by the module IBMBJTT.

On entry from compiled code, register 1 points to the address of
the character-string target. The TIME macro instruction is
issued using the TU parameter. The time is returned in units of
26.04 microseconds and the module converts this into PL/I
defined format 'hhmmssttt' in character format. Under CMS, time
is returned to the nearest second.

DATE

The PL/I DATE built-in function is implemented by module
IBMBJDT.

On entry from compiled code, register 1 points to the address of
the date character string. The TIME macro instruction is
issued. On return, register 1 contains the date in yydddc
packed decimal format. The year is placed in the target
character string in character form. The day of the year is then
compared against a table indicating the number of days in each
month. If the year is a leap year the number of days for
February is set to 29 in the table. The days and months are
then set in the character string and the result returned to
compiled code in the form yymmdd.

DELAY

The PL/I DELAY statement is implemented by calling the DELAY
module IBMBJDY. Register 1 is pointed at the milliseconds delay
required. The milliseconds are converted into units of 26
microseconds and the result stored in a fullword addressed by
the TUINTVL parameter in the STIMER macro instruction. The
STIMER macro instruction is then activated and the delay
started. After the delay, control is returned to the calling
program.

Under CMS, the DELAY statement has no effect.

DISPLAY

The PL/I DISPLAY statement is implemented by the module IBMBJDS.
There are two entry points:

IBMBJDSA   Entry from compiled code.

IBMBJDSB   Entry from IBMBJWT or IBMTJWT when a WAIT for the
            EVENT is reached.

If the parameter list passed to the module has one element, then
the entry is for DISPLAY only, and a VDA is obtained. If there
are two parameters, the entry is for DISPLAY REPLY and a VDA is
again obtained. If there are three parameters, then the entry
is for DISPLAY REPLY EVENT. If the event variable is active,
ERROR is raised. If the event variable is inactive, it is set
active, I/O display and incomplete, and non-LIFO storage is
obtained in which to build the parameter list.

Next the reply buffer, if present, is filled with blanks and, if
the reply string is variable length, its current length is set
to the maximum length. The parameter list to the WTO macro is
now built in the storage obtained, the address of the ECB put
into the event variable if there is one, and a WTO macro issued.
Finally, if DISPLAY REPLY without EVENT was specified, a WAIT
macro is issued for the ECB. Return is then made to compiled
code.

In an MVS environment, PL/I can operate with various sort products, such as OS/VS Sort/Merge, its follow-on, DFSORT, or a program with the same interface. The PL/I programmer can make use of the sort facilities through a call to the built-in subroutine PLISORT. The method of using the facility is fully described in OS PL/I Optimizing Compiler: Programmers' Guide.

The OS/VS sort 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, IBMBKST, is used, and this module acts as an interface between SORT and PL/I. The bootstrap module saves 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, the bootstrap module restores the environment for SORT on return from the exit.

Saving and restoring the environment consists of replacing the address of the error handler in the TCA with the address of an error routine in IBMBKST, and vice versa.

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 that was activated before 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 back chain that omits the SORT DSAs and includes a DSA that is specially flagged so that it can be recognized by the GOTO code. The chaining of save areas is shown in Figure 96 on page 238.

When IBMBKST is called, an area of workspace is acquired by the bootstrap routine IBMBKST. This consists of one level of library workspace, which is flagged and chained to look like two DSAs.

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 a specially flagged DSA in the chain. This is the second save area of IBMBKST. If one is found, a return code of 8 is set up and return made to the SORT routine. If there is a GOTO or an error, then error code 16 is set instead of 8 if the SORT program product being used is that which supports this return code to exits. This results in the termination of the SORT routine, and the GOTO can then be continued in the usual manner by following the DSA back-chain 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 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 96 on page 238. Whichever exit is taken, control passes to this code. The code saves the registers passed by SORT and restores the registers of the bootstrap module IBMBKST, 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.

If there is an error in SORT, control is also passed to code which restores the environment, and passes control to IBMBKST and then to IBMBERR.

Summary of Work Done by the SORT Module

Before calling the SORT program, IBMBKST does the following:

1. Obtains a VDA for two DSAs.

2. Creates a parameter list suitable for SORT.

3. Sets up addressability code for exit routine, if any.

4. Changes the interrupt handler address so that an interrupt 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.

When a SORT E35 or E15 exit is being taken, the addressability code saves the registers of SORT and reestablishes the PL/I environment, and then branches to an entry point of IBMBKST, which:

1. Restores the PL/I interrupt handler address, so that control will pass to the PL/I error-handling routines if a program interrupt occurs.

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 IBMBKST passes to the SORT routines.

```
  ┌─> ┌─────────────────────┐
  │   │ Back-chain          │        ┌──────────────┐
  │   ├─────────────────────┤        │ DSA for PL/I program
  │   │                     │        │ requiring SORT facilities
  │   │                     │        │
  │   └─────────────────────┘        └──────────────┘
  │
  │   ┌─────────────────────┐  <───────┐
  └───┤─Back-chain          │          │  ┌──────────────┐
      ├─────────────────────┤          │  │ Sort bootstrap DSA on
      │ First save area:    │          │  │ calling SORT
      │ for SORT interface  │          │  │
      │ module              │          │  │
      ├─────────────────────┤          │  └──────────────┘
      │ *Exit table         │          │
  ┌─> ├─────────────────────┤          │
  │   │ Back-chain          ├──────────┘  ┌──────────────┐
  │   ├─────────────────────┤             │ Sort bootstrap DSA on
  │   │ Second save area:   │             │ calling exit routine
  │   │ for exit routine    │             │
  │   │ interface           │             │
  │   ├─────────────────────┤             │
  │   │ Work area for the   │             │
  │   │ interface routines  │             └──────────────┘
  │   ├─────────────────────┤
  │   │ Address of SORT     │
  │   │ save area           │
  │   └─────────────────────┘
  │
  │   ┌─────────────────────┐
  └───┤─Back-chain          │             ┌──────────────┐
      ├─────────────────────┤             │ PL/I exit procedure DSA
      │                     │             │
      │                     │             │
      └─────────────────────┘             └──────────────┘
```
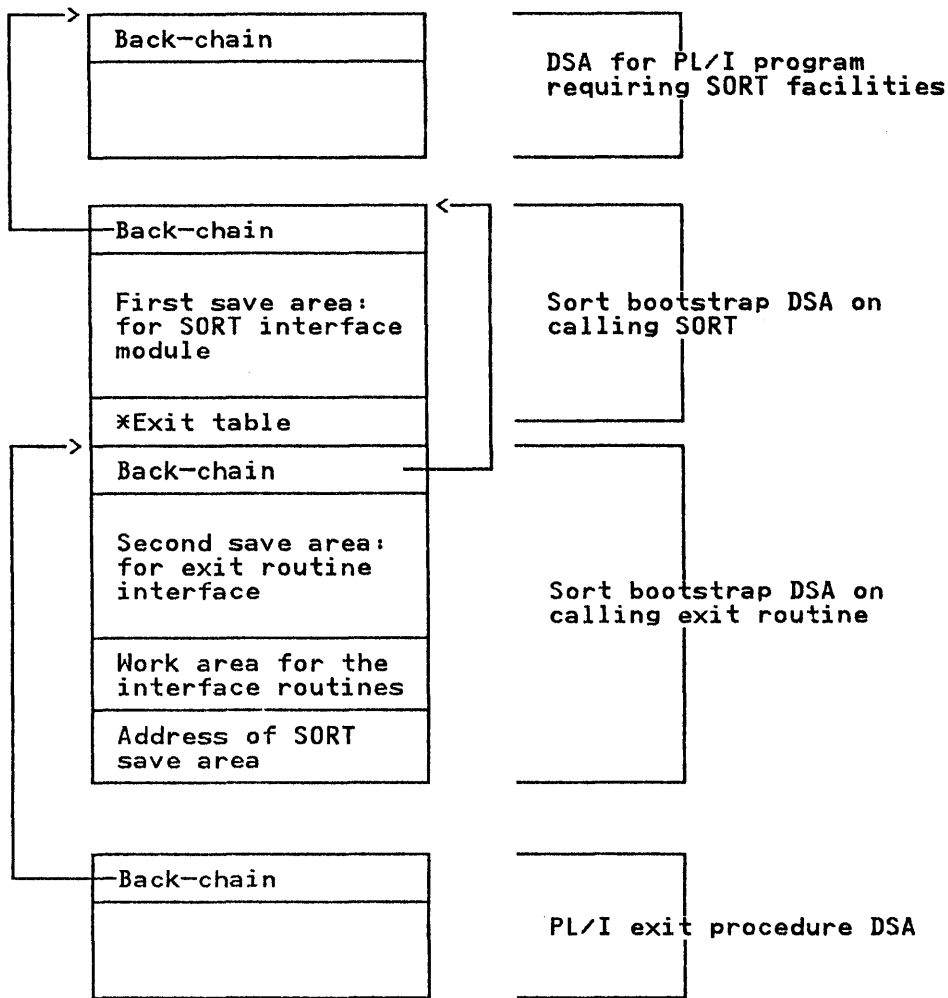
*Exit table

```
┌─────────────────────────────────────────────────────────────────────────────┐
│                                                       ┌                       │
│                           NOP  0                      │ not used              │
│  Entry point for E15      BC   15,12,(15)             │ branch to exit code for E15 exit
│  Entry point for E15      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 bootstap          │ initialized address of routine
│                           DC   A (save area 1)        │ address of first save area
│                                                       └                       │
└─────────────────────────────────────────────────────────────────────────────┘
```

Figure 96.   DSA Chaining during the Execution of SORT

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. These actions are summarized in Figure 97 on page 239.

Main procedure	Call SORT bootstrap Return on completion of SORT
↓ A ↑	
SORT **bootstrap** **IBMBKST**	Set program-check exit for SORT to code in SORT bootstrap. Arrange parameters for SORT. Store registers in first bootstrap DSA. Call SORT.
↓ A ↑	
SORT	Sort as instructed by parameters.
	Save registers in SORT save area. Restore registers for bootstrap. Branch to bootstrap.
↓ A	
Addressability **code**	
↓ A	
SORT bootstrap	
↓ A ↑	
PL/I exit **routine**	Carry out processing — return to SORT bootstrap.

On entry from SORT:
Reset program-check for PL/I.
Set up parameters for exit routine from information passed by SORT.
Call exit routine.

On entry from exit routine:
Reset program-check exit for return to SORT.
Arrange parameters for SORT.
Restore SORT registers.
Return to SORT.

Figure 97. Summary of Action during Use of a SORT Exit

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 IBMBKCP, which issues the CHKPT macro instruction.

In an MVS environment, before the CHKPT 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 workspace acquired by
the module IBMBKCP.

When a restart is made, control is passed to the module IBMBKCP
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 programmers, operating under MVS,
to halt processing until a specified number of events are
complete. The WAIT statement has no effect under VM/CMS.

In the OS PL/I Optimizing compiler, an event can be associated
with either a record I/O operation or a DISPLAY statement. It
can also be an inactive event that is not associated with any
operation or with an attached subtask.

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 (that is, 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.

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 incomplete.
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 event, by use of the COMPLETION pseudo-variable.
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 pseudo-variable.

WAIT Statement (Nonmultitasking)

The wait statement in a nonmultitasking environment is implemented by a call to the resident library routine IBMBJWT. IBMBJWT 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. Waits in a multitasking environment are described in "Communication between Tasks" on page 309.

The WAIT makes use of the OS data-management WAIT macro instruction. However, because of the differences between the facilities offered by the OS and the PL/I language, considerable housekeeping problems are involved for waits on more that 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 OS 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 IOB. 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 illustrated with examples from the skeleton program in Figure 98 on page 242.

```
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 98. Example of WAIT Implementation Problems

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 98, 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 statement. The
entries are termed EVTAB <u>elements</u>. 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 example in Figure 98 on page 242, 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 in 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 pseudo-variable 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 A, "Control Blocks" on page 326.

1. <u>Event variable</u>. 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 is associated with an I/O event, the address of the FCB for the file.

2. <u>ECB (event control block)</u>. Used to hold information about the event at the system level. For I/O events, ECBs are part of the IOB. For DISPLAY events, the equivalent control block is the display control block, which is set up by the display module.

3. <u>EVTAB (event table)</u>. 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. <u>ECB list</u>. 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 WAIT macro instruction.

Wait Module (IBMBJWT)

The actions of the wait module, IBMBJWT, are shown in the flowchart in Figure 99 on page 246, and are described in detail in the publication <u>OS PL/I Resident Library Program Logic</u>.

As the flowchart shows, the WAIT module sometimes issues a WAIT macro instruction, and sometimes relies on the CHECK macro instructions in the PL/I transmitters. The reasons for this are as follows.

The CHECK 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 not economical 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 less than the total number of events in an event list have to be completed, an ECB list is generated for all incomplete events and a WAIT macro instruction is issued.

The WAIT 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
CHECK macro instruction in the transmitter, even though the
events are known to be complete. This is because the CHECK
macro instruction carries out various checking functions as well
as waiting until the event is complete.
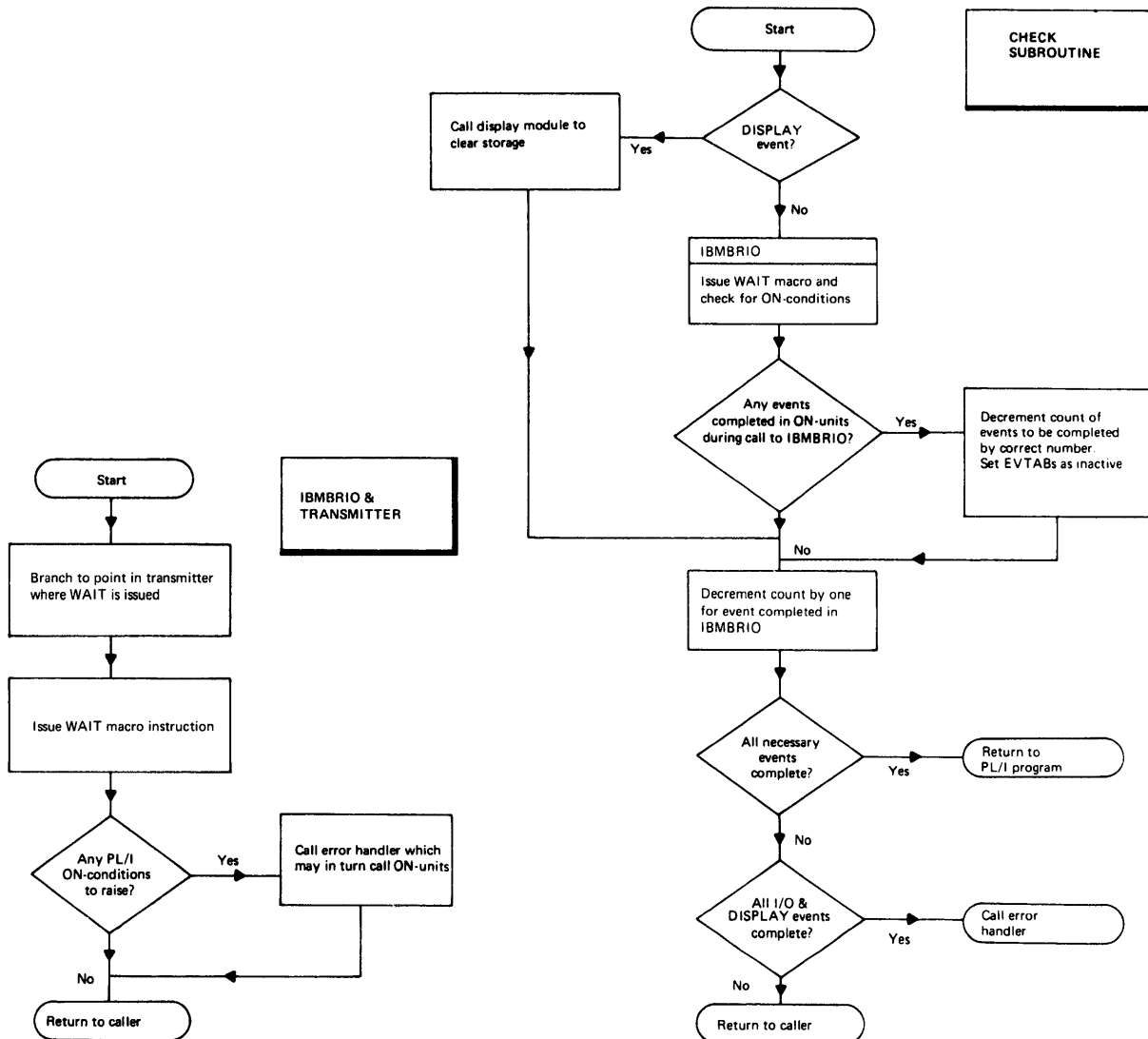
Figure 99 (Part 1 of 2). Summary of the WAIT Statement
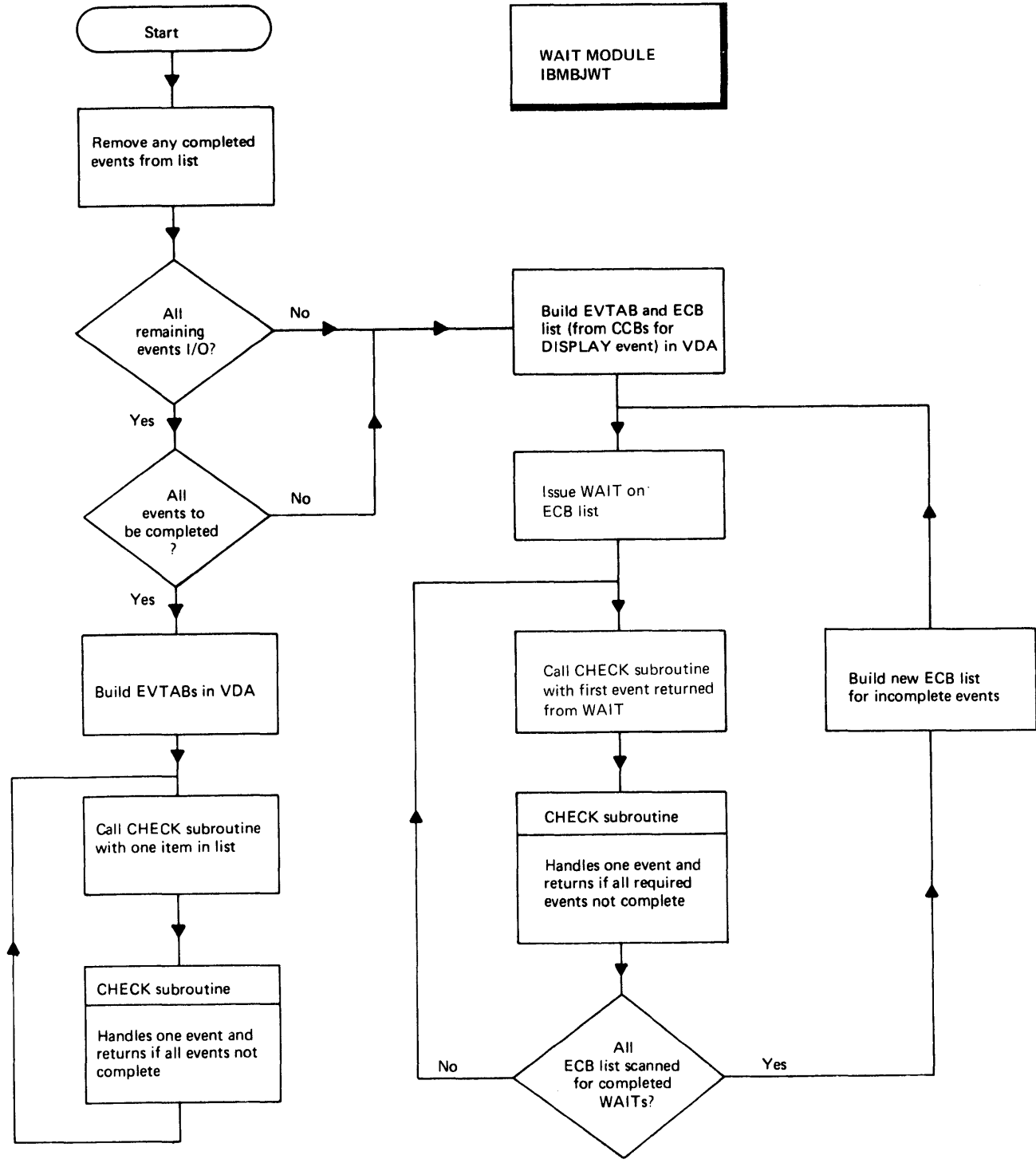
Figure 99 (Part 2 of 2). Summary of the WAIT Statement

CHAPTER 12.   DEBUGGING USING DUMPS

The OS PL/I Optimizing Compiler allows you to obtain an
execution-time dump by calling PLIDUMP.   Using SYSABEND or
SYSUDUMP in the JCL does not normally result in a dump after a
program interrupt or, except in certain exceptional cases, after
an ABEND.   This is because SPIE/ESPIE and STAE/ESTAE routines
result in all interrupts, and the majority of ABENDs, being
passed to the PL/I error handler.

Certain types of program error can, however, result in
overwriting of the control information used by the error
handling routines.   When this occurs, an ABEND will be issued
that results in system action.   This ABEND has a user code of
4000.   Provided that a SYSABEND or SYSUDUMP DD statement was
included in the JCL, an ABEND dump will then be generated.

ABEND dumps are possible under these circumstances.

1.   When an interrupt occurs during the execution of one of the
     error handling routines.

2.   When housekeeping control blocks have been overwritten after
     an ABEND in the program.

3.   If the NOSPIE or NOSTAE option has been used.

4.   An error occurred in the program and the user has coded an
     appropriate IBMBEER module.

The first two of these situations are most probably caused by
overwriting of control information by the PL/I program.   The
first can be identified because a message is sent to the console
that reads 'INTERRUPT IN ERROR HANDLING ROUTINES PROGRAM
TERMINATED', and the ABEND code will be 4000.

Chapter 7, "Error and Condition Handling" on page 105, describes
the methods used to handle interrupts and ABENDs.   It also
describes the implementation of PLIDUMP.   This chapter is
concerned solely with debugging using the facilities provided.

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 within a dump.   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 100 on page 249.

This chapter is divided into four sections:

"Section 1: How to Obtain a PL/I Dump" on page 250

This section 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.
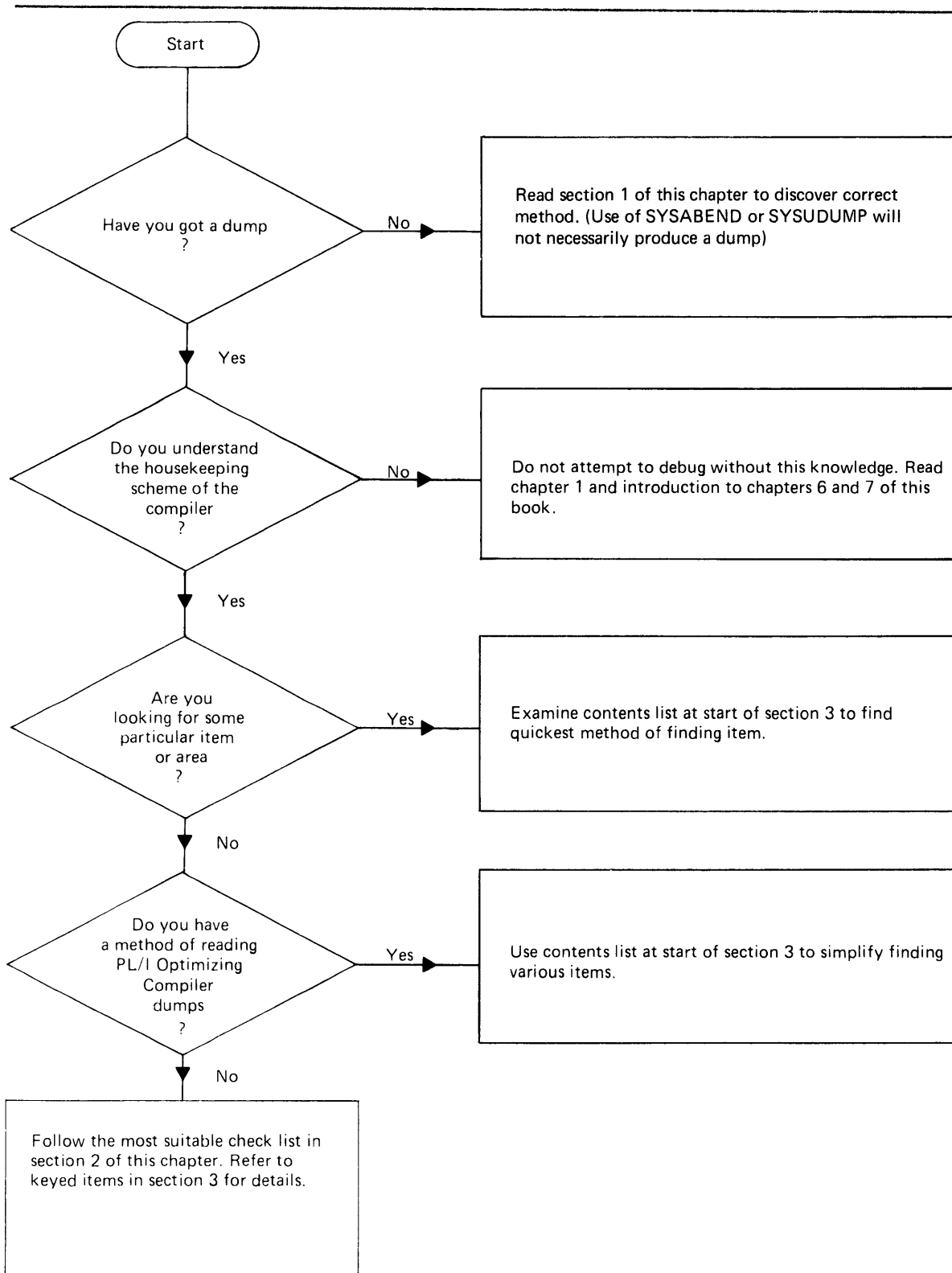
Figure 100.   How to Use this Chapter When Debugging

"Section 2: Recommended Debugging Procedures" on page 258

This section recommends two ways of debugging a PL/I program using a dump. The first shows a PL/I dump that was called from an ERROR ON-unit; the second shows debugging with a system dump that was probably generated because the housekeeping control blocks were overwritten.

"Section 3: Locating Specific Information" on page 263

This section describes how to find various data areas and other information. It is indexed and numbered for quick reference.

"Section 4: Special Considerations for Multitasking" on page 280

This section describes the special considerations that must be taken into account when debugging a program that uses multitasking.

If you are familiar with system dump methods, read the first section before taking a dump. PL/I uses methods that do not follow OS. Use the next two sections when debugging. If you know what you are looking for, go directly to "Contents" on page 263. This section directs you to numbered sections that give details of how to find particular items. If you have no preferred scheme of your own, you can follow the recommended procedures in "Section 2: Recommended Debugging Procedures" on page 258. It cross-refers to the items in "Section 3: Locating Specific Information," so that the details of the steps involved can be quickly found.

SECTION 1: HOW TO OBTAIN A PL/I DUMP

In order to get a formatted PL/I dump, you must include a call to PLIDUMP in your program.

CALL PLIDUMP

The statement CALL PLIDUMP may appear wherever a CALL statement is used. It has the following form:

```
CALL PLIDUMP
    (character-string-expression 1,
     character-string-expression 2);
```

Character-string-expression 1 is a "dump options" character string consisting of one or more of the following dump option characters:

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 SNAP hexadecimal dump of the region 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.

K Produce a hexadecimal dump of the TIOAS and TWA (CICS control blocks) if they exist

NK No dump of CICS blocks.

Tasking Options

A <u>All</u>, which results in a dump of all active tasks including the control task—see Chapter 14, "Multitasking" on page 307.

O <u>Only</u>, which results in a dump of the current task and a dump of the control task.

E <u>Exit</u>, which results in the termination of the task after the dump.

The default options are TFCANHNB. That is, trace information, file information, no block information, no hexadecimal dump, all tasks, and continuation after the information has been put out.

Options are read from left to right. Invalid options are ignored, and, if contradictory options are coded, the rightmost options are taken.

Character-string-expression 2 is a "user 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.

If PLIDUMP is called a number of times in a program, a different user identifier should be used on each occasion. This will simplify identification of the point at which the dump was called.

RECOMMENDED CODING

For PLIDUMP to produce a dump, a DD card for PLIDUMP must be included in the JCL. 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.

By including the statement CALL PLIDUMP ('HB','dump identifier'); in an ERROR ON-unit, 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 101 on page 252.

```
                          (1)      (2)      (3)

               *PROCESS LIST MAP GOSTMT FLOW (n,m);

(4)◄━━┥  (SIZE, SUBSCRIPTRANGE, STRINGRANGE):
         DUMPER: PROC;
         ON ERROR CALL PLIDUMP ('HB', 'ERROR ON-UNIT DUMP');
            ·
            ·       (5)
            ·
         END;
```
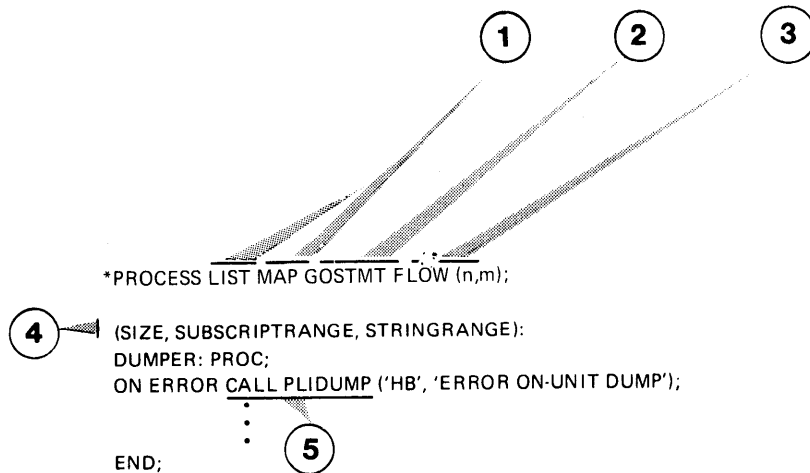
(1) These options give compiled code listing and static storage map, essential for interpreting any dump. MAP results in the generation of a table showing offsets of static and automatic variables from their defining bases.

(2) Permits trace of statement numbers in original source program, and simplifies program checking.

(3) Provides trace of last n branch-out/branch-in points in up to m blocks if SNAP or PLIDUMP with trace is used.

(4) 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.

(5) 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 'TFCANHNB')

Dump identifier character string

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
A	Dump all tasks
O	Dump current task only
E	Exit from task after dump
K	Produce a hexidecimal dump of TIOAS and TWA
NK	No dump of CICS blocks

Printed at head of dump. May be up to 90 characters long.

Figure 101. Code for Debugging

If an ERROR ON-unit containing a call to PLIDUMP is to be
included in an existing program, it is necessary to recompile
the program. This course is advisable as it allows other
diagnostic aids, such as SUBSCRIPTRANGE, to be included.
However, if recompilation is not desirable, a PL/I dump can be
obtained by using a small bootstrap routine that contains an
ERROR ON-unit calling PLIDUMP. This routine can be compiled and
then link-edited with the object module of the program that
needs to be dumped. The ON-unit will then be inherited by the
program that requires a dump, and a dump will be generated when
an error occurs. A suitable bootstrap program is shown in
Figure 102. When using this method, the bootstrap must be
link-edited as the MAIN procedure; it should therefore be passed
to the linkage editor before the program that requires dumping,
since that program will also have the MAIN option. If the
program that requires dumping expects to be passed parameters,
the bootstrap procedure should use an identical parameter list
in its PROCEDURE statement, and should include an identical
argument list in the CALL statement used to invoke the inner
procedure.

```
BOOTSTRAP:  PROC OPTIONS (MAIN);

            DCL program* ENTRY EXTERNAL;

            ON ERROR CALL PLIDUMP ('HB',
            'BOOTSTRAP');

            CALL program*;

            END;
```
*The name of the program to be dumped should be inserted at the
points marked program* in this example.

Figure 102.   Suggested Method of Obtaining a Dump when
              Recompilation is Particularly Undesirable.   (See
              text before using this method.)

If the program that requires dumping already has an ERROR
ON-unit, this will override the ERROR ON-unit in the bootstrap
program.

In certain circumstances, a dump can still be obtained.

1. If the reason for the entry to the ON-unit is the occurrence
    of a PL/I condition, an ON-unit for this condition in the
    bootstrap program will result in a dump being taken before
    the ERROR ON-unit is executed.

    (For example, if the CONVERSION condition was occurring in
    the program to be dumped, a CONVERSION ON-unit could be
    included in the bootstrap program. Such an ON-unit would be
    entered before the ERROR condition was raised.)

2. Provided that the ERROR ON-unit does not include a GOTO out
    of the ON-unit, a FINISH ON-unit can be used. Since the
    standard system action for the ERROR condition is to raise
    the FINISH condition, the dump will be generated after the
    ERROR ON-unit has been executed.

There is no point in including SUBSCRIPTRANGE or other prefixes
in the bootstrap routine, because these are not inherited by
called programs.

The bootstrap method is not recommended unless there are
particularly strong reasons for avoiding recompilation.

CONTENTS OF A PL/I DUMP

The appearance of a typical dump produced by the PLIDUMP modules with the options TFHBA is shown in Figure 103 on page 255. The contents of particular sections follow.

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. The offset of the statement is 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 was specified, the main entry name is given.

   For multitasking programs, the name of the task variable, its status, and the absolute priority of the task are printed. If no task variable is supplied, 'NONE' is printed as the name of the task variable. A dummy task variable will have been supplied (see Chapter 14).

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 (for example, 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 3- or 4-letter abbreviation. A list of these abbreviations is given in Figure 104 on page 256.

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 given.

5. If a hexadecimal dump is requested, 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 traced, the following information is also included:

   a. The address of IBMBERRs DSA

   b. The contents of the general and floating point registers at the time IBMBERR 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

```
                                    * * * PL/I DUMP * * *
    USER IDENTIFIER :   EXAMPLE OF PLIDUMP
                                    * * * CALLING TRACE * * *
        (TCA ADDRESS 00008008 )
PLIDUMP WAS CALLED FROM STATEMENT NUMBER 2 AT OFFSET +00009E FROM A ERR  TYPE ON-UNIT WITH ENTRY ADDRESS 01B0055C
        (AND DSA ADDRESS 000088E8 )
                                            ERROR DIAGNOSTICS
        PL/I CONDITION DETECTED: CONV
            ONCODE   =      612                    SEE LANGUAGE REFERENCE MANUAL
            ONCHAR   =T                             CHARACTER CAUSING CONVERSION ERROR
                     =E3                            AS ABOVE IN HEXADECIMAL
            ONSOURCE =THIS WILL RAISE CONVERSION    STRING CAUSING CONVERSION ERROR
                     =E3C8C9E240E6C9D3D340D9C1C9E2C540C3D6D5E5C5D9E2C9D6D5
                                                    AS ABOVE IN HEXADECIMAL

   ADDRESS OF ERROR HANDLER'S SAVE AREA 00008698
            REGISTERS ON ENTRY TO ERROR HANDLER
        REGS 0-7  00008698      00008690      00008618    81B02C56    00008458    0000859A    00008628    81B003BA
        REGS 8-15 00000001      000085B3      00000000    01B01D02    00008008    00008640    81B02E48    01B02132
                                            END OF ERROR DIAGNOSTICS
   WHICH WAS CALLED FROM A LIBRARY MODULE WITH ENTRY ADDRESS 01B02C50 (AND DSA ADDRESS 00008640 )
   WHICH WAS CALLED FROM A LIBRARY MODULE WITH ENTRY ADDRESS 01B01A98 (AND DSA ADDRESS 00008348 )
   WHICH WAS CALLED FROM A LIBRARY MODULE WITH ENTRY ADDRESS 01B00610 (AND DSA ADDRESS 000085C8 )
   WHICH WAS CALLED FROM STATEMENT NUMBER 6 AT OFFSET +0000B4 FROM PROCEDURE EXAMPLE WITH ENTRY ADDRESS 01B00498
        (AND DSA ADDRESS 000084C8 )

                                    * * * END OF CALLING TRACE * * *

        TRACE OF PL/I CONTROL BLOCKS
        TASK COMMUNICATIONS AREA
ADDR.  OFFSET  0         4         8         C         10        14        18        1C
00008008 00000  00000000 000088B8 00008008 00000000 00000000 00008018 00005FB0 00008210 ..............................
00008028 00020  00000000 00008298 00008138 000082A0 00000000 00008268 00000000 00000000 ..............................
00008048 00040  00008230 00000000 800064FC 00000000 00006228 00000000 00000000 00000000 ..............................
00008068 00060  00000000 01B01920 01B019B4 800064BC 800064BE 80006500 01B02132 F0000B0B ...........................0...
00008088 00080  582E0004 58EE0000 19DF478C 00C21851 181F180E 58FC00F0 05EF9500 C001470C ...................B.......0...
000080A8 000A0  00BA18E5 58FC00AC 07FF07FE 01B016C0 000058FC 0078051F D80118E0 18D19834 ...V......................J..
000080C8 000C0  D0209140 D001078E 9140D001 478C00DC D203D04C D0509120 D001078E D201D056 ...-..... ......K.........K...
000080E8 000E0  D0549180 D054071E 181F58FC 00F407FF 000080B2 00000000 000080B2 000080B2 .................4.............
00008108 00100  000080B2 000080B2 00000000 00000000 00000000 00000000 00000000 00000000 ..............................
00008128 00120  000080B2 0000000C                                                       ........
        TCA IMPLEMENTATION APPENDAGE
                                    * * * PL/I DUMP * * *
ADDR.  OFFSET  0         4         8         C         10        14        18        1C
00008138 00000  00046800 00000000 01B02048 00001070 00000000 00000000 00000000 00000000 ..............................
00008158 00020  00008248 000082F0 01B01876 00000000 00000000 00000000 00005EC8 00000000 .......0................;H....
00008178 00040  01B018CC 01B0294C 00000000 00000000 00008008 00000000 81B00250 01000030 ..............................
00008198 00060  00046220 00000000 00000000 00001000 00000000 00000000 00000000 00000000 ..............................
        LIBRARY WORK SPACE
CONTENTS OF REGISTER SAVE AREA
        REGS 0-7  800077E0     01B003E0     000088E8     81B014AE     00008458     00000000     01B003E0     01B00420
        REGS 8-15 000089AA     000089C8     000084C8     00008138     00C00000     00008780     81B014EE     800077FE

ADDR.  OFFSET  0         4         8         C         10        14        18        1C
00008780 00000  08000110 000088E8 00000000 81B014EE 800077E0 800077FE 01B003E0 000088E8 .......Y.....................Y
000087A0 00020  81B014AE 00008458 00000000 01B003E0 01B00420 000089AA 000089C8 000084C8 ........................H...H
000087C0 00040  00008138 00000000 00008808 000089D0 00020000 000087F8 00440000 00100000 ..........................8.....
000087E0 00060  F0F0F3F2 F1F20000 00000000 0000006C 80007AC4 F0C2F45C 40404040 40404040 003212............D0B4*
00008800 00080  40404040 40404040
        DYNAMIC SAVE AREA (ON-UNIT)
CONTENTS OF REGISTER SAVE AREA
        REGS 0-7  000089D0     01B003E0     81B005B2     01B00340     00008458     00000000     00008580     01B00420
        REGS 8-15 000089AA     000089C8     000084C8     00008138     00000000     000088E8     81B005FA     01B014A8

ADDR.  OFFSET  0         4         8         C         10        14        18        1C
000088E8 00000  8C248928 00008698 050C0000 81B005FA 01B014A8 000089D0 01B003E0 81B005B2 ..............................
00008908 00020  01B00340 00008458 00000000 00008580 01B00420 000089AA 000089C8 000084C8 ... ....................H...H
00008928 00040  00008138 00000000 00008780 000089D0 000089D0 91E091E0 000084C8 00000000 ........................H....
00008948 00060  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 ..............................
00008968 00080  00000000 00000000 00008928 00000000 00000000 00000000 00000000 00000000 ..............................
00008988 000A0  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 ..............................
000089A8 000C0  0000C2E2 000089AA 00020000 0000C5E7 C1D4D7D3 C540D6C6 40D7D3C9 C4E4D4D7 ..BS..........EXAMPLE OF PLIDUMP
000089C8 000E0  000089B6 00120000                                                       ........
        DYNAMIC SAVE AREA (LIBRARY)
```

Figure 103.   An Example of PLIDUMP

Abbreviation	Condition Name
AREA	AREA
CHCK	CHECK
COND	CONDITION (programmer named 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 104. Abbreviations for Condition Names Used in PLIDUMP
Trace Information

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.

3. The contents of the FCBs, DCBs, DCLCBs, IOCBs, and exclusive
   file blocks are given in formatted hexadecimal notation, if
   either the 'H' or 'B' option is also included.

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'17'		–
X'18'		–
X'19'	PENDING	100
X'1A'	ATTENTION	400
X'CD'		9250
X'CF'	ERROR	1000
X'D3'		9200
X'D5'		3500
X'D7'		4050
X'D9'		5050
X'DF'		5000
X'E1'	ERROR	9050
X'E3'		1000
X'E5'		4000
X'E7'		xxxx
X'E9'		4050
X'EB'	ERROR	0003
X'ED'		1000
X'EF'	ERROR	1550
X'F1'		1500
X'F3'		2000
X'F5'		3768
X'F7'	ERROR	3000
X'F9'	ERROR	3800
X'FB'		3900
X'FD'		9000
X'FF'		8090

Figure 105.   Error Code Field Lookup Table

Hexadecimal Dump

The hexadecimal dump is produced by the execution of a SNAP macro instruction. Thus the normal SNAP dump is produced.

It should be noted that the PSW will contain the address of an instruction in IBMBKMR, one of the modules used to implement PLIDUMP. This will bear no relation to the error in the dumped program.

If the program is not multitasking, the SNAP macro specifies all register save areas, subpools, task control blocks, and, provided the O (Only) option is not included in the PLIDUMP options, the trace table.

For a dump of a multitasking program the contents are:

In the control task

    Register save areas
    Subpools
        Trace table
        Control blocks

In the other tasks

    Register contents
    Register save areas
        Subpools
        Jobpack Area
        Linkpack Area

Block Option

When the block option is used, the contents of the TCA, the TIA (TCA appendage), 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; that is, library DSA, procedure DSA, ON-unit DSA, or dummy DSA. The contents of the FCBs, DCLCBs, and IOCBs for any open files are printed in a similar format.

In a dump of a multitasking program, the contents of the tasking appendage are also printed.

If the option A(all) is used in a multitasking program, the TCA, TIA, DSAs and tasking appendage of all directly ascending tasks will be printed. FCBs, IOCBs, DCLCBs will be printed after files open in any task if the option A is used.

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 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 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: Locating Specific Information" on page 263, refer to the key areas shown above. The object program listing allows you to study the instructions that are being carried out and to find various control blocks in static storage. The linkage editor map allows you to identify particular parts of the executable program phase and to identify the routine associated with each DSA. The object program listing is produced by the LIST compiler option; the linkage editor map, by MAP.

Note: The PSW in the SNAP dump should not be consulted. This will give the address at which the SNAP macro instruction was issued. This is an address in one of the PLIDUMP modules and is not relevant to the error in the problem program. Instead, look at the trace information.

DEBUGGING OVERLAID STORAGE

Storage overlay is one of the most common errors you usually debug with a dump. In PL/I applications, overlay problems can be divided into these categories:

- Are you using a subscript outside the declared bounds (SUBSCRIPTRANGE)?

- Did you attempt to assign a string to a target with a shorter maximum length (STRINGSIZE)?

- Does one of the arguments to a SUBSTR reference fail to comply with the rules described for the SUBSTR built-in function (STRINGRANGE)?

- Were significant high-order (left-most) binary or decimal digits lost during an assignment to a variable, an intermediate result, or on an input/output operation (SIZE)?

- Are you reading a variable-length file into a variable?

- Are you using a pointer variable?

By understanding these problem areas before you proceed through the dump, you can isolate the problem much faster.

The first four categories are associated with the indicated PL/I conditions, all of which are disabled by default. If you suspect one of these problems is in your program, use the appropriate condition prefix on the suspected statement or on the BEGIN or PROCEDURE statement that defines the block that contains the suspected statement.

The fifth category occurs when you read a data record into a variable that is too small. This type of problem only happens with variable-length files, and can often be isolated by examining the data in the file information, and the data in the buffer.

The last category occurs when you misuse a pointer variable. This type of storage overlay is particularly difficult to isolate.

There are a number of ways pointer variables can be misused:

1. When a READ statement with the SET option is executed, a value is placed in a pointer. If you then execute a WRITE statement or another READ SET option with another pointer, you will overlay your storage if you try to use the original pointer.

2. When you attempt to use a pointer to allocated storage that has already been freed, you can also cause a storage overlay.

3. When you attempt to use a pointer, set with the ADDR built-in function, as a base for data with different attributes, you can cause a storage overlay.

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 only a system ABEND 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. However, in such cases, the main storage situation can be investigated by following the methods itemized in "Section 3: Locating Specific Information" on page 263.

Throughout each of the two recommended procedures given in the following paragraphs, there are cross-references to the methods given in "Section 3: Locating Specific Information." The cross-references consist of the keys by which the methods are identified; for example, H6, D5. These keys are listed in "Housekeeping Information in All Dumps" on page 263.

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 is available at the head of the dump. An error message is generated, which provides a useful starting point. First, examine the type of the error and the point where it occurs. Next, examine the ONCODE and other condition built-in function values, along with the trace information. We suggest the following procedure:

1.   Examine the error by means of the ONCODE and any other relevant built-in function values. These values are given in the trace information. (The meanings of codes are given in the <u>OS and DOS PL/I Optimizing Complier Language Reference Manual</u>).

2.   Find the location of the error (P1 on page 263) and the block in which the error occurred (H12 on page 264). If the error occurred in a library module, see H14 on page 264. This information is normally available from the head of the PLIDUMP in the trace information.

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 and H9). Distinguish between compiled code and library register usage.

7.   If SUBSCRIPTRANGE or STRINGRANGE is not enabled, check that the error was not caused by one of these conditions.

8.   Check housekeeping (H1-H16) starting with the area most directly concerned with type of statement in which the error occurred.

9. Check values of all variables in the program (V1-V6).

10. Check the logic of code being executed from object listing.

System ABEND Dump

Provided a SYSABEND or a SYSUDUMP card is included in the JCL, a system ABEND dump will be generated when there is a failure of the error-handling modules, or of the module that prints the PL/I hexadecimal dump. 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.

Because ABENDs caused by overrunning the specified time (SYSTEM 322) do not enter the STAE/ESTAE exit, these will cause dumps to be generated in normal circumstances.

An ABEND 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 results in a dump.

Thus, an ABEND is produced if:

- The program interrupt exit was reset during the program.

    This exit is normally set by the program initialization routines to prevent a dump.

- The program interrupt exit was never set at all.

    This possibility is extremely unlikely.

- The program check exit itself is not working, and the SPIE/ESPIE 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 was reset by the program. The program interrupt exit is always reset for the duration of error handling or PLIDUMP, to prevent looping should an interrupt occur. (For further details, see Chapter 7, "Error and Condition Handling" on page 105.)

If an interrupt occurs during error handling, an ABEND with a code of 4000 is produced. This results in a dump if SYSABEND or SYSUDUMP cards were provided. 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 was overwritten during the execution of the program.

The most practical solution may be to recompile the program with SUBSCRIPTRANGE, STRINGSIZE, and STRINGRANGE enabled. Then rerun the program with the NOSPIE and NOSTAE execution-time options. These PL/I conditions check for possible overwriting by subscripts or substrings that are beyond the bounds of the variable referred to.

If a 4000 ABEND must be run, execute with NOSPIE and NOSTAE. For more information on 4000 ABEND codes, see <u>OS PL/I Optimizing Compiler: Debug Guide</u>.

However, having obtained an ABEND dump, the following debugging procedure may be adopted.

1. Determine whether the dump was caused by an interrupt in the error-handling routines or a housekeeping error discovered during the analysis of an ABEND. If the cause was an interrupt in the error handler, a message will have been sent to the console before the ABEND was issued, and the ABEND will have a code of 4000, if the interrupt occurred in one of the error-handling routines. Note that codes 322 and 122 may also give system dumps, and that the use of NOSPIE or NOSTAE can result in the generation of a dump.

2. Locate the instruction causing the interrupt. This is done by looking for the PSW (O1).

3. Inspect this instruction to see if it appears to have been overwritten, bearing in mind the cause of the interrupt; for example,

    a. Do the registers used in the instruction contain incorrect information, picked up because of overwriting?

    b. Is it a branch to a protected address?

4. Inspect the TCA(O5) to ensure that all error-handling addresses are correct.

5. Investigate the housekeeping fields, starting with the DSA chain (H1-H3), then the chain of ONCAs (H5,H6).

6. Investigate the error that caused entry into the error handler. This can be done by examining the contents of IBMBERR's DSA (H7) and the associated ONCA (H6). See whether incorrect information passed to the error handler could be causing a failure.

7. Check for uninitialized variables (particularly pointers), and incorrect passing of parameters.

8. If none of the above produces a solution, an error in the error-handling modules is a possibility. If you decide to call IBM for assistance at this point, see "Appendix B: Requirements for Problem Determination and APAR Submission" in OS PL/I Optimizing Compiler: Programmer's Guide. The cause of the original entry to the error handler may already be known, and can perhaps be avoided by altering the source program so that the error does not occur. 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 one of the routines it calls, the highest probability is still that the program check exit was altered in the error handler and that an invalid branch was then made from one of the addresses in the TCA because of overwriting. Therefore, you should carefully check the TCA. (See "Task Communication Area (TCA)" on page 407 for a map of the TCA.) If this fails to produce results, return to stage 2 of the above procedure.

SECTION 3: LOCATING SPECIFIC INFORMATION

This section tells you how to find information in a dump. The section is organized in modular form for easy reference. You should look through the list below to discover the items in which you are interested. Suggested methods of debugging a PL/I program from a dump are given in "Section 2: Recommended Debugging Procedures" on page 258. Unless you are experienced in using dumps, or are looking for some particular item, use the procedures in "Section 2: Recommended Debugging Procedures," rather than attempting to find various items through the information in this section.

CONTENTS

Key Areas of a PL/I Dump

P1   Statement number and address where error occurred (dump called from ON-unit only)

P2   Type of error (dump called from ON-unit only)

P3   Register contents at time of error or dump invocation

P4   The DSA chain

P5   The TCA

P6   Timestamp

Key Areas of an ABEND Dump

O1   Finding address of interrupt

O2   Type of interrupt

O3   Register contents at point of interrupt

O4   The DSA chain

O5   The TCA

O6   Find the program interrupt element (PIE) or extended program interrupt element (EPIE)

Stand-Alone Dumps

S1   Finding key areas in stand-alone dumps

Housekeeping Information in All Dumps

H1   Following the DSA back-chain

H2   Associating instruction with correct module

H3   Following calling trace

H4   Associating DSA with block

H5   Finding relevant ONCA

H6   Following the chain of ONCAs

H7   Finding information from IBMBERR's DSA

H8   Finding and interpreting register save areas

H9   Register usage

H10   Following ISA free-area chain

H11   Finding the task variable

H12   Block structure of program (static back-chain)

H13   Forward chain in DSAs

H14   Action if error is in a library module

H15   Discovering contents of parameter lists

H16   Finding main procedure DSA

H17   Finding the relationships between tasks

H18   Finding the tasking appendage

H19   Finding the TCA from the tasking appendage

H20   Following the heap free-area chain

H21   Following the heap storage chain

Finding Variables

V1   Automatic variables

V2   Static variables

V3   Controlled variables

V4   Based variables

V5   Area variables

V6   Variables in areas

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 ABEND 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.

Note: If PLIDUMP is called a number of times in a program, a different user identifier should be used with each CALL statement so that the point at which the dump was taken is obvious.

P2: Type of Error (Applies to 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 ONCODE 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 not been generated, the contents of the registers can be found from the save area in the DSA. 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.

For a dump called from an ON-unit, the method of finding the register contents is as follows:

1. Find the DSA of IBMBERR. The value of register 13 will be found in the chainback field at offset 4 of this DSA.

2. If the interrupt was a program check interrupt (see Figure 106 on page 266), 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 IBMBERR's DSA in the order 14 through 11. See Figure 106 on page 266.

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 interrupts, 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 IBMBERR's DSA. This is set to 0 for program check interrupts, and to 1 for other interrupts.
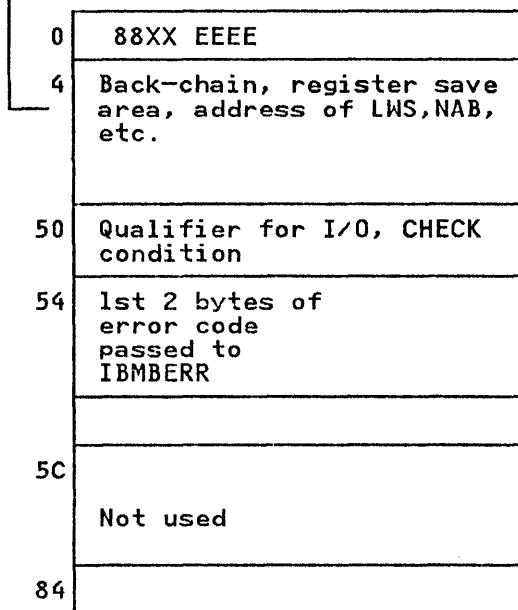
FINDING REGISTER VALUES IF INTERRUPT OCCURRED IN LIBRARY ROUTINE: If the ON-unit was entered from a library module, a search back through the DSA chain to the first compiled code DSA should be made. This can be discovered from the trace information or by following the back-chain from IBMBERR's DSA (offset 4 in each DSA) until a procedure block, begin block, or ON-unit DSA is found. This may be determined from flag bits 4 and 5 of DSA, as follows:

Bit 4	Bit 5	DSA
0	0	Procedure block
1	0	Begin block
1	1	ON-unit

Software detected interrupt

DSA of block in which interrupt occurred

Offset	Content
0	
4	Back-chain
C	
	Registers 14 through 11 at the time of interrupt
44	Other DSA information

DSA for IBMBERR

Offset	Content
0	88XX EEEE
4	Back-chain, register save area, address of LWS,NAB, etc.
50	Qualifier for I/O, CHECK condition
54	1st 2 bytes of error code passed to IBMBERR
5C	Not used
84	

Program check interrupt

DSA of block in which interrupt occurred

Offset	Content
0	
4	Back-chain
8	
C	Interrupt address from word 2 of PSW
14	Registers 0 through 11 at time of interrupt
44	Other DSA information

DSA for IBMBERR

Offset	Content	
0		
4	Address of interrupt DSA	
8	Register save area, address of LWS, NAB, etc.	
54	Error code created by IBMBERR	
58		interrupt code
5C	Register 14 at time of interrupt	
60	Register 15 at time of interrupt	
68	Floating point registers 0, 2, 4, 6	

Floating point registers are saved only if interrupt relates directly to a PL/I condition, and return may be made to the point of interrupt

Figure 106. The Contents of IBMBERR's DSA After a System Detected and a PL/ I Interrupt

The value of register 12 can only be discovered in a DSA prior to a compiled code DSA, as it is not stored by library routines when they are entered. 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 in 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 back-chain can then be followed to find the
DSA for IBMBERR. The DSA for IBMBERR can be recognized if an
ON-unit is involved, because it will be the DSA before the
ON-unit DSA. IBMBERR's DSA is always headed by a flag word of
hexadecimal Y'8800EEEE', meaning that it is a library DSA in
LIFO storage. To identify IBMBERR's DSA for certain, register
15 of the previous block's DSA must be inspected to see if it
points to the module IBMBERR.

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 (library workspace).

See H1 and Figure 107 for details of how to follow the DSA
chain.
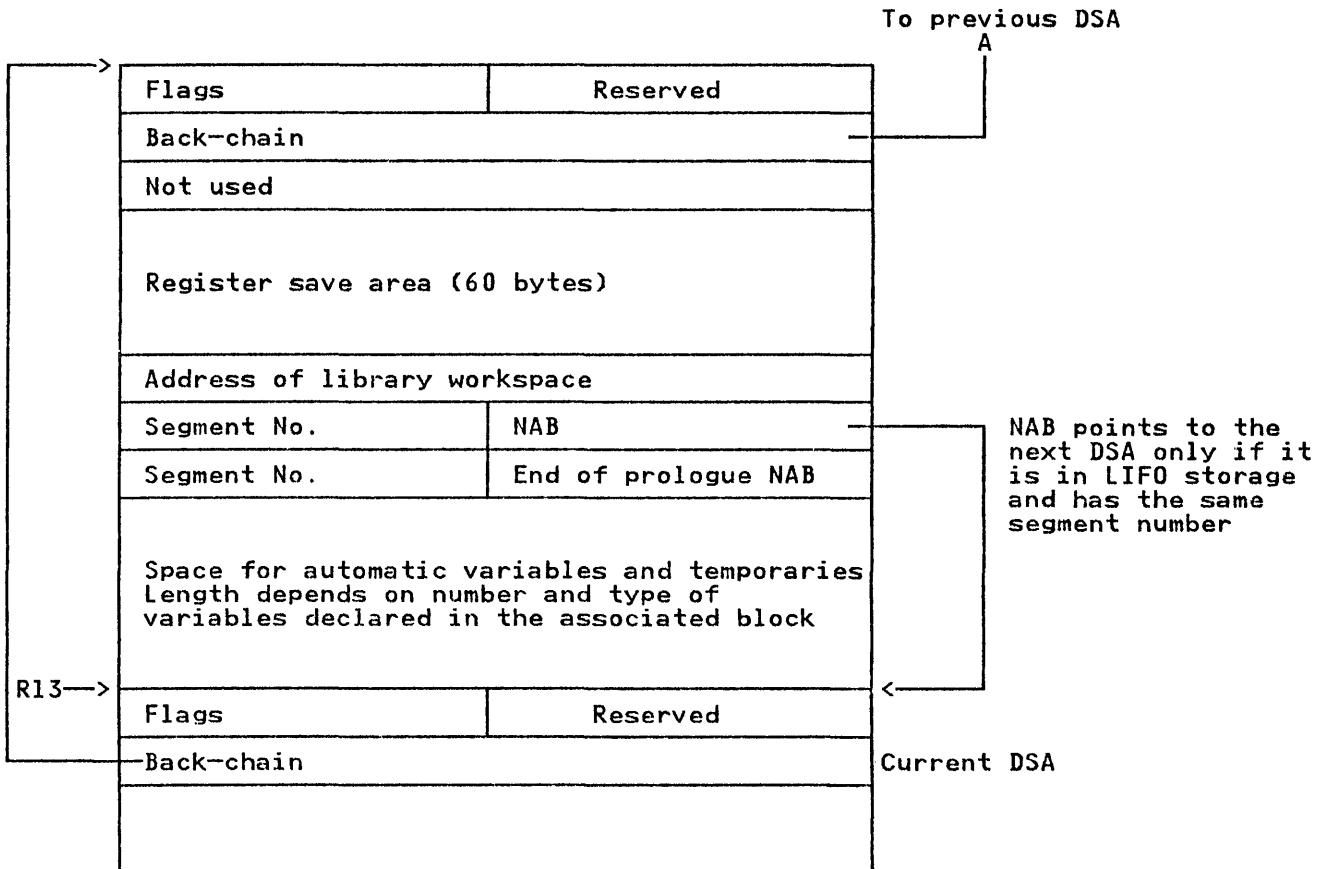
Figure 107. The Chaining of DSAs

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.

The TCA is addressed by register 12. The format of the TCA is given in "Task Communication Area (TCA)" on page 407. The use of the various fields is explained in Chapter 4, "Communication between Routines" on page 64. If NOTRACE is specified, the TCA is in subpool 1, preceded by the characters ZTCA.

P6: Timestamp

If the TSTAMP installation option is specified in your installation, the date and time of compilation are in the last 16 bytes of the static control section. The first word gives the offset to the information. The static control section is addressed by register 3. If the BLOCK option is specified, the timestamp is printed at the head of the static blocks.

KEY AREAS OF AN ABEND DUMP

01: Address of Interrupt

If the ABEND code is 4000, 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. The PSW is held in subpool 5. ABENDs are discussed further in the OS and DOS Optimizing Compiler: Debug Guide.

The associated statement number in the source program can normally be found by finding the last compiled code DSA, and finding the point at which the exit was made (register 14 in the save area). The address of the program control section in the link-edit map can then be subtracted from this address; the offset compared to the listing gives the appropriate statement number.

Finding the statement number is not likely to prove useful because of the circumstances in which a 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. If the ABEND code is not 4000, see "06: Finding the Program Interrupt Element (PIE/EPIE)" on page 269.

02: Type of Interrupt

The type of interrupt can be found from the first word of the PSW (see Principles of Operation for details).

03: Register Contents at the Point of Interrupt

Registers 14 through 2 appear in the PIE (program interrupt element). Registers 3 through 13 are those printed in the save area trace. See 06 for finding the PIE.

O4: 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 108 on page 273.

O5: The TCA

Register 12 should point at the TCA.

O6: Finding the Program Interrupt Element (PIE/EPIE)

The program interrupt element (PIE) or extended program interrupt element (EPIE) is found in subpool 5. The PIE/EPIE is followed by registers 3 through 13 and then the STAE/ESTAE work area. The STAE/ESTAE work area holds the last problem program PSW.

This is the value required for finding the original cause of the ABEND if the ABEND code is other than 4000.

STAND-ALONE DUMPS

S1: Finding Key Areas in Stand-Alone Dumps

The programmer should attempt to find the various PL/I key areas (TCA, DSA chain, etc.) discussed above. See the debugging manual for your operating system.

HOUSEKEEPING INFORMATION IN ALL DUMPS

H1: Following the DSA Back-chain

Each DSA holds a back-chain 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 back-chain 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.)

For programs using multitasking, the DSA back-chain leads to the dummy DSA of the major task. The DSA of the block in which the task was attached is not included in the chain. To find this DSA, the 'static' back-chain held at offset X'58'(88) can be used provided the procedure attached as a task is internal to the attaching block. If the procedure is not internal, the NAB value X'4C'(76) in the DSA before it will normally point to the required DSA.

(The method of chaining during a multitasking program is explained in Chapter 14, "Multitasking"). For relationship of NAB and DSA chaining, see H13.)

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, the programmer 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; if 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 under "Using the Table of Offsets" on page 271. 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 (in a compiled code DSA, flag bits 4 and 5 are 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 associated 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 step 1 or step 2. 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; 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'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 third 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.
    Following this offset, the statement number is given in
    2-byte hexadecimal format.

PROCEDURE NAME: To find the entry point name, a back-chain 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 DSAs 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 of 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 that, 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 is for IBMBERR. The first byte of the
error code is held in this DSA at offset 84 (X'54'). Compare
this byte with the values in Figure 105 on page 256. This error
code is given an associated PL/I condition. It is the ON-unit
for this condition that is 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, "Interlanguage Communication" on page 281). If this
is possible, chain back one more DSA and try again. (To check
if this has occurred, see step 3 on page 270.)

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 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 finding the entry point
name (see "H2: Associating Instruction with Correct" on
page 269), it is possible to follow the calling trace.

H4: Associating DSA with Block

DSAs are associated with code by finding the register values in
the preceding DSA register save area (H8) 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. The block in the source
program can be identified by statement numbers or entry point,
described in "H2: Associating Instruction with Correct" on
page 269.

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 IBMBERR (follow back the DSA
    chain until register 15 in the save area points to IBMBERR).
    See H1, H3, H7. If this is a library DSA (flag bits 4 and 5
    set to '10') go to 3, below.

2. Find the LWS addressed from this DSA. The address 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 library DSA in LWS.

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 IBMBERR's DSA

The information held in IBMBERR's DSA is used by the error
message modules for information about the error. If the
messages have not been generated, the information can be deduced
from the DSA. The contents of IBMBERR's DSA are shown in
Figure 106 on page 266. See H4 for associating DSAs with
correct code. IBMBERR's DSA can be identified by X'EEEE' in
bytes 2 and 3.

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 108. Each DSA holds the register values as they were on exit from its block.

H9: Register Usage

Register usage is fully discussed in Chapter 2, "Compiler Output" on page 12. A summary of register usage, showing which registers are always used for a particular purpose, is given in Figure 109 on page 275.

H10: Following the ISA Free-Area Chain

The ISA 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 X'1C' (28) 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.
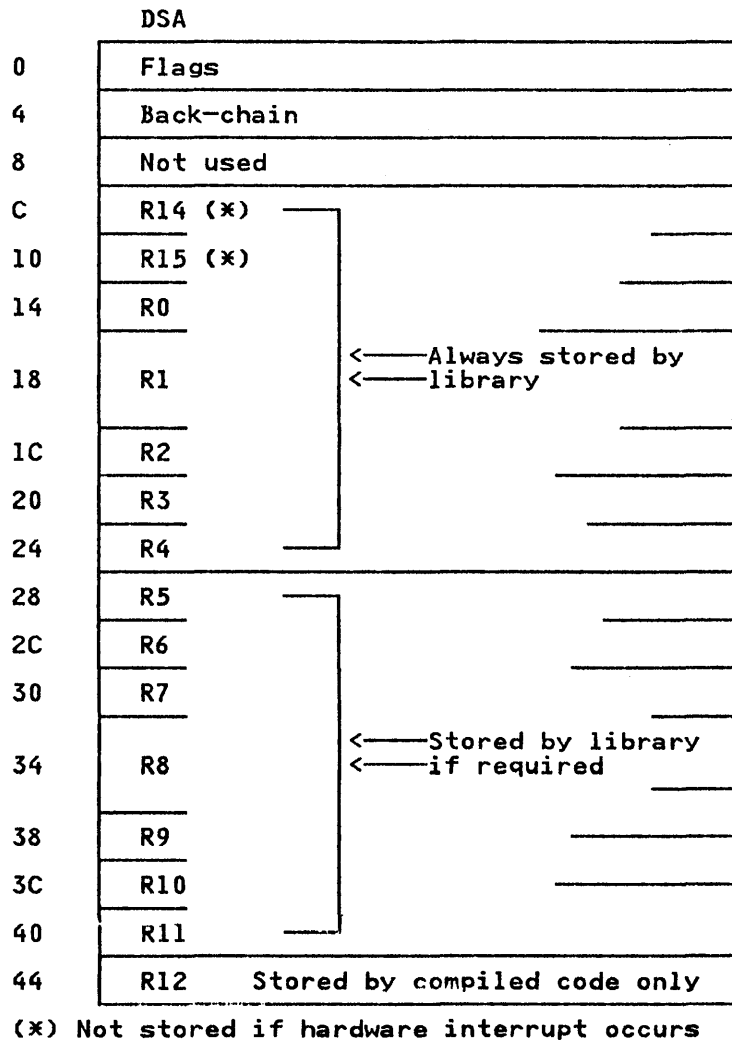
DSA

Offset		
0	Flags	
4	Back-chain	
8	Not used	
C	R14 (*)	
10	R15 (*)	
14	R0	
18	R1	<———Always stored by <———library
1C	R2	
20	R3	
24	R4	
28	R5	
2C	R6	
30	R7	
34	R8	<———Stored by library <———if required
38	R9	
3C	R10	
40	R11	
44	R12 Stored by compiled code only	

(*) Not stored if hardware interrupt occurs

Figure 108. The Register Save Area in the DSA

H11: Finding the Task Variable

The task variable is held in the TCA at offset X'24'(36).

H12: Block Structure of Program (Static Back-chain)

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 encompassing block. The chain thus formed is known
as the static back-chain.

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 '00' 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

The fact that the interrupt or the error was discovered during
the execution of a library module suggests that a check must be
made on the data that is being passed to the module.

To discover the contents of a parameter list, see H15.

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: Associating DSA with Block"
on page 272) of the calling block. Register 1 will then locate
the parameter list. If the list is in static storage, this can
be compared with the static storage listing. The name of the
called routine can be discovered (H3). The correct parameters
for PL/I library routines are given in the appropriate library
Program Logic Manual.

H16: Finding Main Procedure DSA

The main procedure DSA can be found by following the back-chain
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. NAB
is held at offset X'4C'(76) in the dummy DSA. The address of
the dummy DSA is held at offset X'24'(36) 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
and the character value ZDSA before it.

Register	Compiled Code Usage	Library Usage
R0	Work register	Work register
R1	Work register	Work register
R2	Program base[1],[2]	Work register[3]
R3	Static base[2]	Program base[2]
R4	Work register	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[2]	TCA pointer[2]
R13	Current DSA pointer[2]	Current DSA pointer[2]
R14	Branch register	Branch register
R15	Link register	Link register

Figure 109. Normal Register Usage

Notes to Figure 109:

[1] The contents of the program base register are saved during in-line record I/O and TRT instructions.

[2] Dedicated register, that is, the contents remain unchanged throughout the execution of the associated compiled code or library routine.

[3] File Control Block (FCB) pointer during record I/O.

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 IBMBERR are slightly different from normal if the interrupt was a hardware interrupt. See Figure 106 on page 266 for a diagram of IBMBERR's DSA.

H17: Finding the Relationship between Tasks

The relationship between tasks can be discovered from the chains in the tasking appendage. The chain held at offset X'28'(40) points to the tasking appendage of the most recently attached subtask.

The chain at offset X'24'(36) points to the task with the same attaching task that was attached before the task being inspected (elder sibling). If there is no such task, the field is set to zero.

The chain at offset X'20'(32) points to the subsequently attached task with the same attaching task (younger sibling).

If there is no younger sibling, this chain points to an offset
within the tasking appendage of the parent task. An attempt to
continue along the chain results in a zero field being met.
(See Figure 126.)

TO FIND THE PARENT TASK: Search along the chain held at offset
X'20'(32) in each tasking appendage. When this field is zero,
the tasking appendage of the parent task has been reached. The
start of this tasking appendage is at an offset of X'-8'(-8)
from the address held in the pointer of the previous tasking
appendage. (See Figure 126.)

TO FIND ALL SUBTASKS OF A TASK: The address of the most
recently attached subtask is held at offset X'28'(40) in the
tasking appendage. Other subtasks can be found by following the
chain held at offset X'24'(36) in the tasking appendage until a
zero field is reached. This will be the end of the chain and is
the first of the active subtasks to be attached by the task.
(See Figure 126.)

TO FIND SIBLING TASKS: Previously attached sibling tasks (elder
sibling) can be found by following the chain held at offset
X'24'(36) in the tasking appendage.

Subsequently, attached sibling tasks (younger siblings) can be
found by following the chain held at offset X'20'(32) in the
tasking appendage. When a zero field in this chain is reached,
the parent task has been found. The most recently attached
sibling task is the last one whose chain field does not hold a
zero value. The word after the zero value will point to the
tasking appendage of this task.

The method used for chaining tasks is explained in
Chapter 14, "Multitasking" on page 307 (See also Figure 126 on
page 314).

H18: Finding the Tasking Appendage

The address of the tasking appendage is held at offset X'2C'(44)
in the TCA and at offset X'50'(80) in the dummy DSA of the
attaching task.

H19: Finding the TCA from the Tasking Appendage

The TCA is addressed from X'2C'(44) in the TCA tasking
appendage.

H20: Following the heap free-area chain

The heap free-area chain connects the areas of heap storage that
are available to satisfy ALLOCATE requests. Heap allocation is
further described in "Allocating and Freeing Heap Non-LIFO
Storage" on page 92.

The chain starts at offset X'78'(120) 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.

H21: Following the heap storage chain

The heap storage chain connects all areas obtained by GETMAIN
macro instructions for use as heap storage. Heap storage is
further described in "Allocating and Freeing Heap Non-LIFO
Storage" on page 92. The chain starts at offset X'74'(116) 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.

FINDING VARIABLES

The value of the variables in the program at the point of
interrupt can be discovered by using 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 a block, see "H4: Associating
DSA with Block" on page 272.)

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 a block, see "H4: Associating DSA with Block" on
page 272.)

V3: Controlled Variables

As described in chapter 2, controlled variables are addressed by
an anchor word that is held in the pseudo-register vector. This
anchor word can be identified from compiled code, while the PRV
offset can be found in the dump. The address of the controlled
variable must be obtained from the PRV in the dump because it is
not filled-in until the ALLOCATE statement is executed.

The address in the pseudo-register vector is the address of the
data or, in certain circumstances, of a descriptor or a
locator/descriptor. These fields are described in
Appendix A, "Control Blocks" on page 326. The data is preceded
by a control block—the controlled variable control block. The
address of the previous allocation is held at an offset of -8
from the address in the PRV. If there is no previous
allocation, the address is set to zero.

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 chapter (see index) or from Appendix A.

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"

Back-chain
    DSA back-chain
    ONCA back-chain
	offset X'4' in DSA
	offset X'0' in ONCA

BOS Beginning of segment	Offset X'8' from TCA
Controlled variables	see "Variables"
DCLCB Declare control block	Deduced from object program listing
DCB	addressed from offset X'14'(20) in FCB
ENVB Environment block	offset X'C'(12) in DCLCB
DED Data element descriptor	deduced from object program listing
Diagnostic statement table	addressed from offset X'8' from entry point of main procedure
DFB Diagnostic file block	addressed from offset X'40'(64) in TCA
DSA Dynamic storage area	addressed by register 13 (see P3 and D3)
EOS End of segment	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 File control block	identified in PL/I dumps. Addressed via PRV and DCLCB
Flow statement table	addressed from offset X'4C'(76) in TCA
Filename	addressed from offset X'10'(16) in FCB
ISA Free-area chain	offset X'1C' (28) in implementation-defined appendage, which is addressed from offset X'28'(40) in TCA
Heap free-area chain	offset X'78' (120) in the implementation-defined appendage
Heap storage chain	offset X'74' (116) in the implementation-defined appendage
Locator/descriptor	deduced from object program listing
LWS Library workspace	addressed from offset X'48'(72) in every DSA
NAB Next available byte	offset X'4C'(76) in DSA
ONCA ON-communications area	the offset of the associated ONCA is held in a halfword at offset X'2' in each section of LWS
ONCB ON-control block start of dynamic ONCB chain	offset X'60'(96) in DSA
first static ONCB	offset X'5C'(92) in DSA
ON-cells	addressed from offset X'70'(112) in DSA
OCB Open control block	deduced from object program listing and parameter list of open module, IBMBOCL
Parameter lists	object program listing and static storage map
Register values	See P3 and O3
RCB Request control block	object program listing and static storage map
SIOCB Stream I/O control block	object program listing
Symbol table	Static listing
Symbol table vector	Static listing
Statement number table	see diagnostic statement table
Static storage	addressed by register 3 in compiled code. See P3 and O3.
Segment number	first two bytes of BOS, or NAB. '00'=1, 'FF'=2, etc.[1]
Tasking appendage	addressed from X'2C'(44) in the TCA.
Task variable	addressed from X'24'(36) in the TCA.

TCA Task communications area	addressed by register 12. See P3 and D3.
Variables automatic	offset from DSA of block in which they are declared. As shown in variables offset map. 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	PRV offset referenced in compiled code holds latest allocation of the variable. A back-chain through the previous allocation can be made using the header chain. See V3.
static	offset from register 3 is shown in variable 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.

[1]   Except when the first two bytes of NAB are filled with
zeros, the first two bytes of BOS are always less than the
first two bytes of NAB when a segment needs to be freed.
For a full discussion of the use of these segment pointers,
and their exceptions, see Chapter 6, "Storage Management" on
page 84.

SECTION 4: SPECIAL CONSIDERATIONS FOR MULTITASKING

The major difference between a dump of a multitasking program
and the dump of any other PL/I program is that certain relevant
items are held within the control task. For this reason, the
control task is always dumped as well as the current task.

The contents of the dump of a tasking program depend on the dump
options specified. If A (all) is used, all the tasks will be
dumped. If O (only current task) is specified, the control task
and the current task will be dumped.

The dump is carried out within the control task and this
prevents access to the tasking housekeeping during the execution
of the dump. However, this does not prevent access by other
tasks to PL/I variables which may be dumped. Subtasks of the
current task can access and alter values within the ISA of the
current task. Consequently, the values of the variables printed
cannot be guaranteed to be those that were current at the
invocation of the dump.

As explained in "Multitasking Housekeeping" on page 311, the DSA
chaining differs slightly when a program is multitasking. The
back-chain passes through the dummy DSA of the task and ends at
the dummy DSA of the major task. The DSA of the block in which
the task was attached is not included in the back-chain.

Compiled code and the static control sections generated by the
compiler are always held in storage associated with the control
task.

CHAPTER 13.   INTERLANGUAGE COMMUNICATION

The OS PL/I Optimizing Compiler allows subroutines compiled on 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 when passing arguments to assembler language routines. These are described under "ASSEMBLER Option" on page 305.

A full description of how to use the interlanguage communication facilities is given in the OS PL/I Optimizing Compiler: Programmer's Guide. A detailed description of the PL/I library routines involved is given in the OS PL/I Resident Library: Program Logic.

This chapter explains the basic design principles of PL/I interlanguage communication. It explains the inner workings of main storage during the execution of a program involving interlanguage calls.

The interlanguage facilities are summarized below for background information.

Summary of Interlanguage Facilities

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. PL/I can call COBOL that calls PL/I that calls FORTRAN; FORTRAN can call PL/I that calls COBOL, and so on. 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. Options also allow the programmer to specify whether he/she wishes data aggregates to be automatically reformatted when passed as arguments. (The programmer may wish to carry out the reformatting himself/herself.)

The language involved is fully described in the language reference manual. Briefly, it is 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 in a COBOL or FORTRAN routine, 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 PROCEDURE or ENTRY statement. To override the creation or remapping of dummy arguments for aggregates, the options NOMAP, NOMAPIN, and NOMAPOUT 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 the other language. The use of the COBOL option in the ENVIRONMENT attribute is described in the last section of this chapter.

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 different environment required for each language. This consists of:

   a. Different methods of handling program checks and consequently a requirement for the issuing of new SPIE/ESPIE macro instructions when a new language is entered.

   b. The dependence of PL/I and FORTRAN on initialization and termination routines to set up and discard their environments.

The first of these problems you must solve yourself by ensuring that arguments passed between the routines are of suitable data types. Arguments and parameters are discussed in detail in the OS PL/I Optimizing Compiler: Programmer's Guide.

The other problems mentioned above are handled automatically by the interlanguage communication facilities of the compiler. They are summarized below.

DIFFERENCES IN DATA AGGREGATES

Structures in PL/I and COBOL, and arrays in PL/I and FORTRAN, are organized in different manners.

COBOL structures are mapped as they are declared, with the structure starting on a doubleword boundary and each item separately aligned. PL/I structures are mapped in a manner that minimizes padding.

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. COBOL data with the OCCURS option, which can be equivalent to PL/I arrays, is held in row-major order, as are PL/I arrays.

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 to be passed. The correct type of parameter list must, therefore, be set up when an interlanguage call is made.

DIFFERENCES OF ENVIRONMENT

PL/I, COBOL, and FORTRAN all have different methods of handling program checks. PL/I allows the programmer to handle all program checks. FORTRAN allows the programmer to handle certain program checks. COBOL leaves program checks almost entirely in the hands of the system. Because of the different requirements, PL/I interlanguage communication must issue a new SPIE/ESPIE macro instruction whenever control passes between languages. The INTER option demands that program checks are analyzed when they occur and that they are passed to the appropriate language. If they are to be passed to PL/I, the PL/I environment must be restored. For these reasons, the INTER option demands that further SPIE/ESPIE macro instructions be issued.

IBM FORTRAN and COBOL 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 SPIE/ESPIE macro instruction to initiate the FORTRAN error-handling scheme. In PL/I, the initialization routines prepare for the PL/I error-handling schemes and also prepare the way for dynamic storage allocation. During PL/I initialization routines, 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 of last-in, first-out dynamic storage, various other housekeeping fields, and storage for variables declared automatic. (See Chapter 1, "Introduction" on page 1, and Chapter 5, "Object Program Initialization" on page 74, for a discussion of the PL/I environment.)

When PL/I is called from either COBOL or FORTRAN 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, and the PL/I environment saved so that it may be restored on return to PL/I.

THE PRINCIPLES OF INTERLANGUAGE COMMUNICATION

Figure 110 on page 284 shows the method used to handle interlanguage communication problems.

Interface code is inserted immediately before and immediately after the execution of a routine in a different language. This code calls the interlanguage communication routine to save the existing environment and to set up the required environment.

Where necessary it creates dummy aggregate arguments of the correct format. The interface code is divided between compiled code and library routines. Compiled code handles data aggregate arguments and calls a library routine to handle the problems of environment. Three PL/I resident library routines are used; one for calls to each language. These routines are known as the interlanguage housekeeping routines.

The interface code is always placed in PL/I, because it is the PL/I compiler that manages the interlanguage facilities. However, the position of the code depends on whether PL/I is the called or calling program.
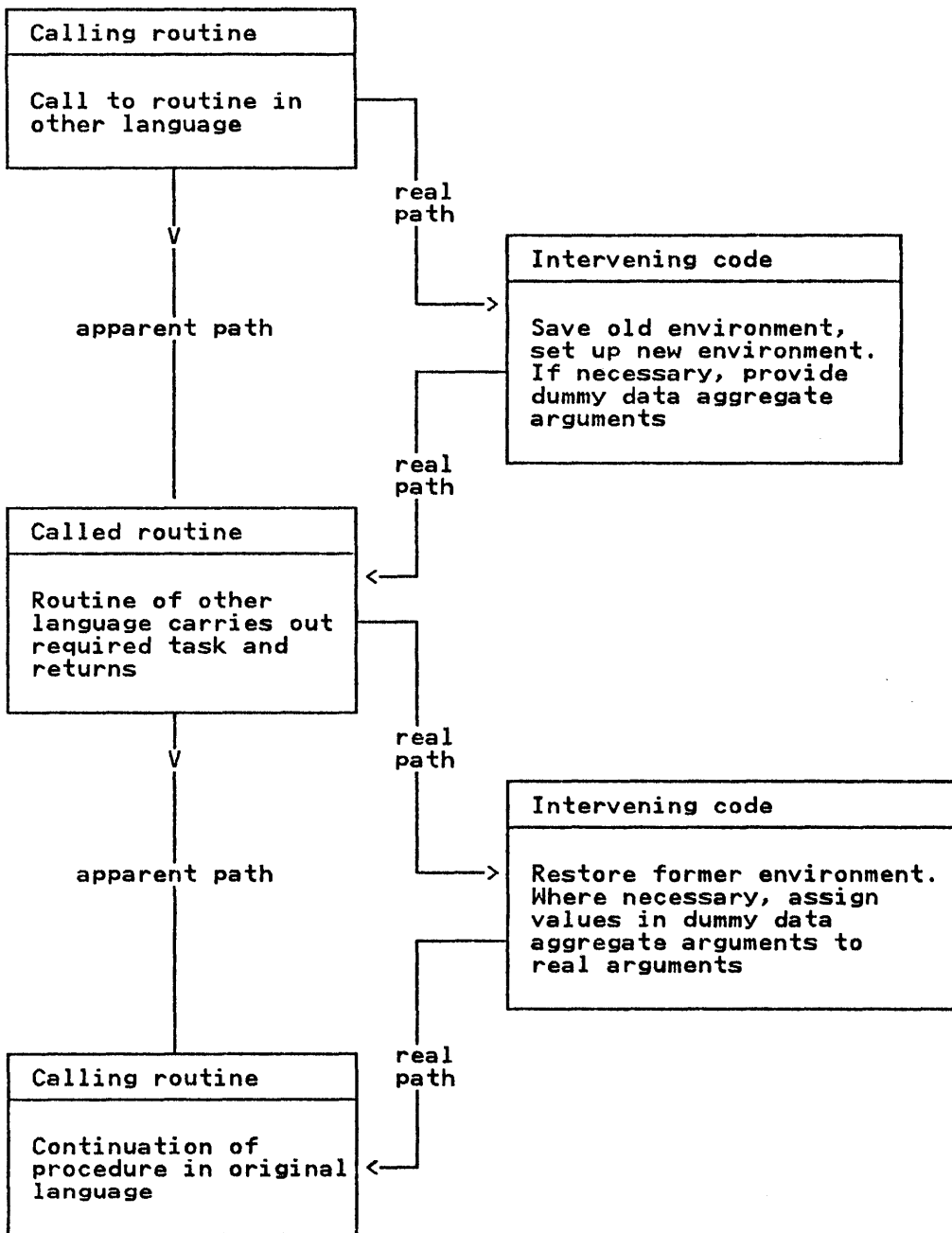
```
┌─────────────────────────────┐
│ Calling routine             │
├─────────────────────────────┤
│                             │
│ Call to routine in          │──────┐
│ other language              │      │  real
│                             │      │  path
└──────────────┬──────────────┘      │
               │                     │    ┌──────────────────────────────────┐
               │                     │    │ Intervening code                 │
               V                     │    ├──────────────────────────────────┤
               │                     └──> │                                  │
               │                          │ Save old environment,            │
    apparent path                         │ set up new environment.          │
               │                     ┌────│ If necessary, provide            │
               │                     │    │ dummy data aggregate             │
               │                     │    │ arguments                        │
               │               real  │    │                                  │
               │               path  │    └──────────────────────────────────┘
┌──────────────┴──────────────┐      │
│ Called routine              │      │
├─────────────────────────────┤      │
│                             │ <────┘
│ Routine of other            │
│ language carries out        │──────┐
│ required task and           │      │  real
│ returns                     │      │  path
│                             │      │
└──────────────┬──────────────┘      │    ┌──────────────────────────────────┐
               │                     │    │ Intervening code                 │
               V                     │    ├──────────────────────────────────┤
               │                     └──> │                                  │
               │                          │ Restore former environment.      │
    apparent path                         │ Where necessary, assign          │
               │                     ┌────│ values in dummy data             │
               │                     │    │ aggregate arguments to           │
               │                     │    │ real arguments                   │
               │               real  │    │                                  │
               │               path  │    └──────────────────────────────────┘
┌──────────────┴──────────────┐      │
│ Calling routine             │      │
├─────────────────────────────┤      │
│                             │ <────┘
│ Continuation of             │
│ procedure in original       │
│ language                    │
│                             │
└─────────────────────────────┘
```

Figure 110.   The Principles of Interlanguage Communication

PL/I Calls COBOL or FORTRAN

When the _calling_ program is PL/I the interface code is placed
immediately before and immediately after the call to the COBOL
or FORTRAN routine.   The sequence is shown in Figure 111 on
page 285, and is summarized below.

1.   Compiled code remaps data aggregate arguments if necessary.

2.   Compiled code calls the interlanguage housekeeping routine,
     which handles environment problems.

3.   Compiled code calls the COBOL or FORTRAN routine.

4. On return from the COBOL or FORTRAN routine, compiled code calls the interlanguage housekeeping routine to restore the PL/I environment.

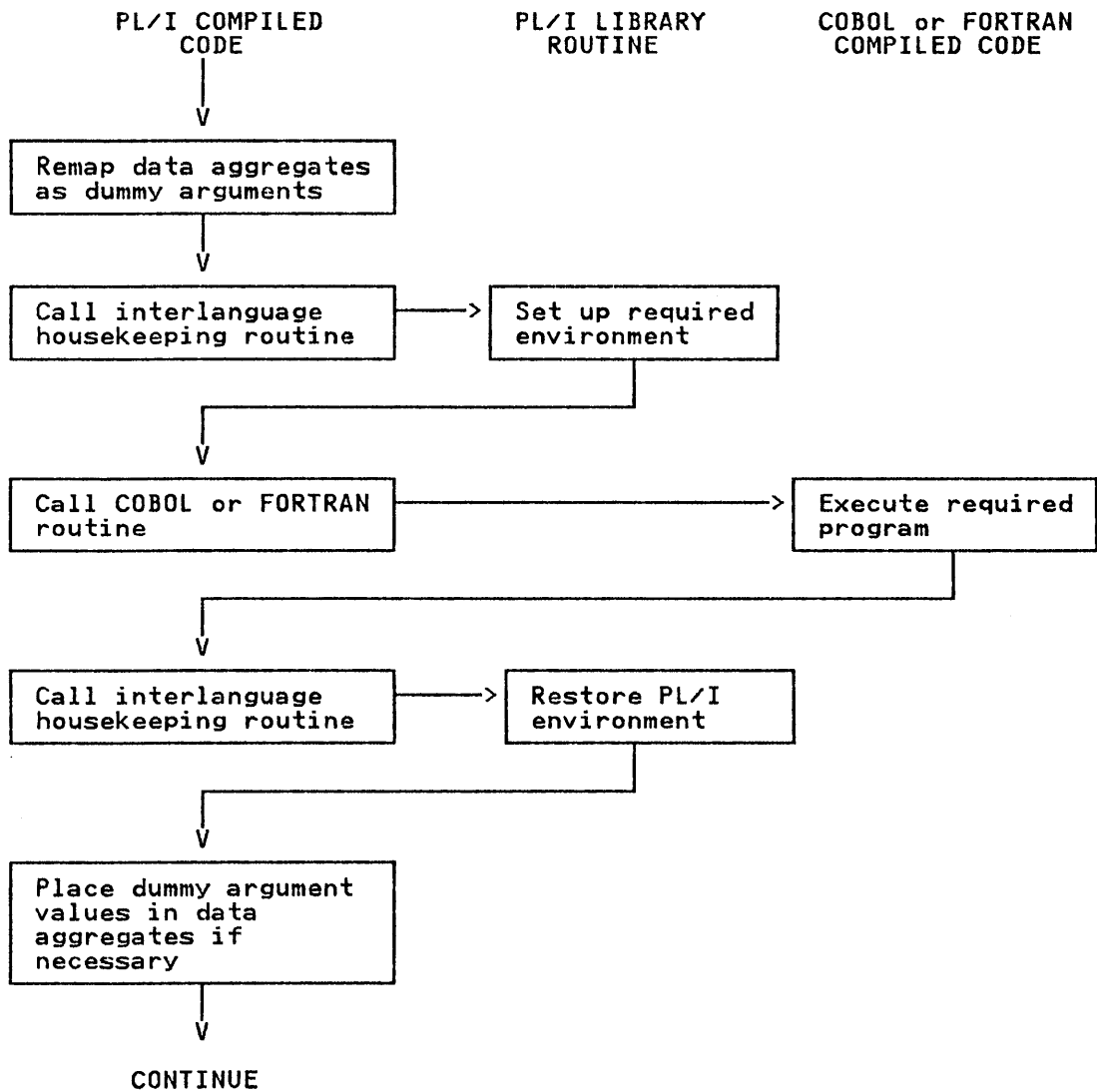5. Compiled code remaps dummy data aggregate arguments if any, and continues.

```
PL/I COMPILED          PL/I LIBRARY         COBOL or FORTRAN
     CODE                 ROUTINE            COMPILED CODE
       |
       V
+--------------------+
| Remap data aggregates |
| as dummy arguments |
+--------------------+
       |
       V
+--------------------+        +------------------+
| Call interlanguage |------->| Set up required  |
| housekeeping routine |      | environment      |
+--------------------+        +------------------+
       |
       V
+--------------------+                              +------------------+
| Call COBOL or FORTRAN |-------------------------->| Execute required |
| routine            |                              | program          |
+--------------------+                              +------------------+
       |
       V
+--------------------+        +------------------+
| Call interlanguage |------->| Restore PL/I     |
| housekeeping routine |      | environment      |
+--------------------+        +------------------+
       |
       V
+--------------------+
| Place dummy argument |
| values in data      |
| aggregates if        |
| necessary           |
+--------------------+
       |
       V
   CONTINUE
```

Figure 111. Calling Sequence When PL/I Calls COBOL or FORTRAN

The code generated by the compiler is shown in Figure 112.

```
                    SOURCE LISTING
    STMT LEV NT

      1    0    P13P2: PROC;
      2  1  0          DCL FRED OPTIONS(COBOL),
                         1 STRUCTURE,
                           2 C CHAR (1),
                           2 D FIXED BINARY (31,0);
      3  1  0          CALL FRED(STRUCTURE);
      4  1  0          END;
```

```
* STATEMENT NUMBER  3
000066   D2 03 D 04C D 050       MVC    76(4,13),80(13)       ┐
00006C   41 00 0 008             LA     0,8(0,0)                Get VDA for
000070   58 10 D 04C             L      1,76(0,13)           ─dummy
000074   1E 01                   ALR    0,1                    arguments
000076   55 00 C 00C             CL     0,12(0,12)
00007A   47 D0 2 01E             BNH    CL.4
00007E   58 F0 C 048             L      15,72(0,12)
000082   05 EF                   BALR   14,15
000084              CL.4         EQU    *                    ─┐ Place new value in
000084   50 00 D 04C             ST     0,76(0,13)           ─ NAB
000088   41 11 0 000             LA     1,0(1,0)
00008C   50 10 D 0B8             ST     1,184(0,13)
000090   18 71                   LR     7,1                  ─Move structure
000092   D2 00 7 000 D 0C3       MVC    0(1,7),STRUCTURE.C    into dummy
000098   58 70 D 0B8             L      7,184(0,13)
00009C   58 90 D 0C4             L      9,STRUCTURE.D
0000A0   50 90 7 004             ST     9,4(0,7)             ─┐ Branch to
0000A4   58 F0 3 014             L      15,A..IBMBIECAA      ─interlanguage
0000A8   05 EF                   BALR   14,15                ─ housekeeping routine
0000AA   58 70 D 0B8             L      7,184(0,13)
0000AE   41 40 7 000             LA     4,0(0,7)
0000B2   50 40 3 03C             ST     4,60(0,3)            ─Set up argument list
0000B6   96 80 3 03C             OI     60(3),X'80'
0000BA   1B 55                   SR     5,5
0000BC   41 10 3 03C             LA     1,60(0,3)            ─┐
0000C0   58 F0 3 040             L      15,64(0,3)           ─Branch to COBOL
0000C4   05 EF                   BALR   14,15                ─ routine
0000C6   58 F0 3 018             L      15,A..IBMBIECCA      ─Branch to ILC house-
0000CA   05 EF                   BALR   14,15                ─ keeping routine
0000CC   58 70 D 0B8             L      7,184(0,13)          ─Move values from
0000D0   D2 00 D 0C3 7 000       MVC    STRUCTURE.C(1),0(7)   dummy to real
0000D6   58 90 7 004             L      9,4(0,7)             arguments
0000DA   50 90 D 0C4             ST     9,STRUCTURE.D        ─┘
```

Figure 112. Code Generated When PL/I Passes a Structure to a COBOL Routine

FORTRAN or COBOL Calls PL/I

When the _called_ program is PL/I, the necessary interface code is
placed at the start and finish of the PL/I program. The
interface code is compiled as an encompassing routine to the
required PL/I routine.

The method used is to compile the PL/I program in the normal
way, except that it is compiled as internal to an interface
procedure that contains the interface code.

This interface, or encompassing, procedure is given the external
name of the PL/I procedure and is thus called by the
other-language routine. The interface procedure, when it has
called the interlanguage housekeeping routine and handled the

data aggregate arguments, calls the required PL/I routine. Control returns to the original caller by way of the interface routine, which again handles the interlanguage problems before returning.

The sequence of events when PL/I is the _called_ program is shown in Figure 113 and is summarized below.

1. A COBOL or FORTRAN routine calls the PL/I routine.

2. Control passes to the interface routine, which has been compiled with the ESD name of the PL/I routine or entry point.

3. The interface routine calls the library interlanguage housekeeping routine to handle environment problems.

4. The interface routine handles data aggregate arguments as necessary.

5. The interface routine calls the compiled program's required routine.

6. Control returns from the required routine to the interface routine. The interface routine handles data aggregate arguments as necessary.

7. The interface routine calls the library interlanguage housekeeping routine to handle environment problems.

8. Control returns from the interface routine to the original caller.

Retaining the Environment

The overhead of setting up PL/I and FORTRAN environments every time a routine is called could become considerable if the routine were called a large number of times. To prevent this overhead, the environment is retained until the routine that calls the other-language routine is itself terminated. The termination is done by a rearrangement of the save area chaining, so that the PL/I and other language termination routines are not entered until the calling routine is itself terminated.

The arrangement introduces certain housekeeping problems, which are resolved by inserting further save areas into the chain. These save areas have register 14 values that result in control being passed to subroutines of the interlanguage housekeeping routines. These subroutines, known as _tail_ code, handle problems such as preserving values passed from the caller to the caller's caller.
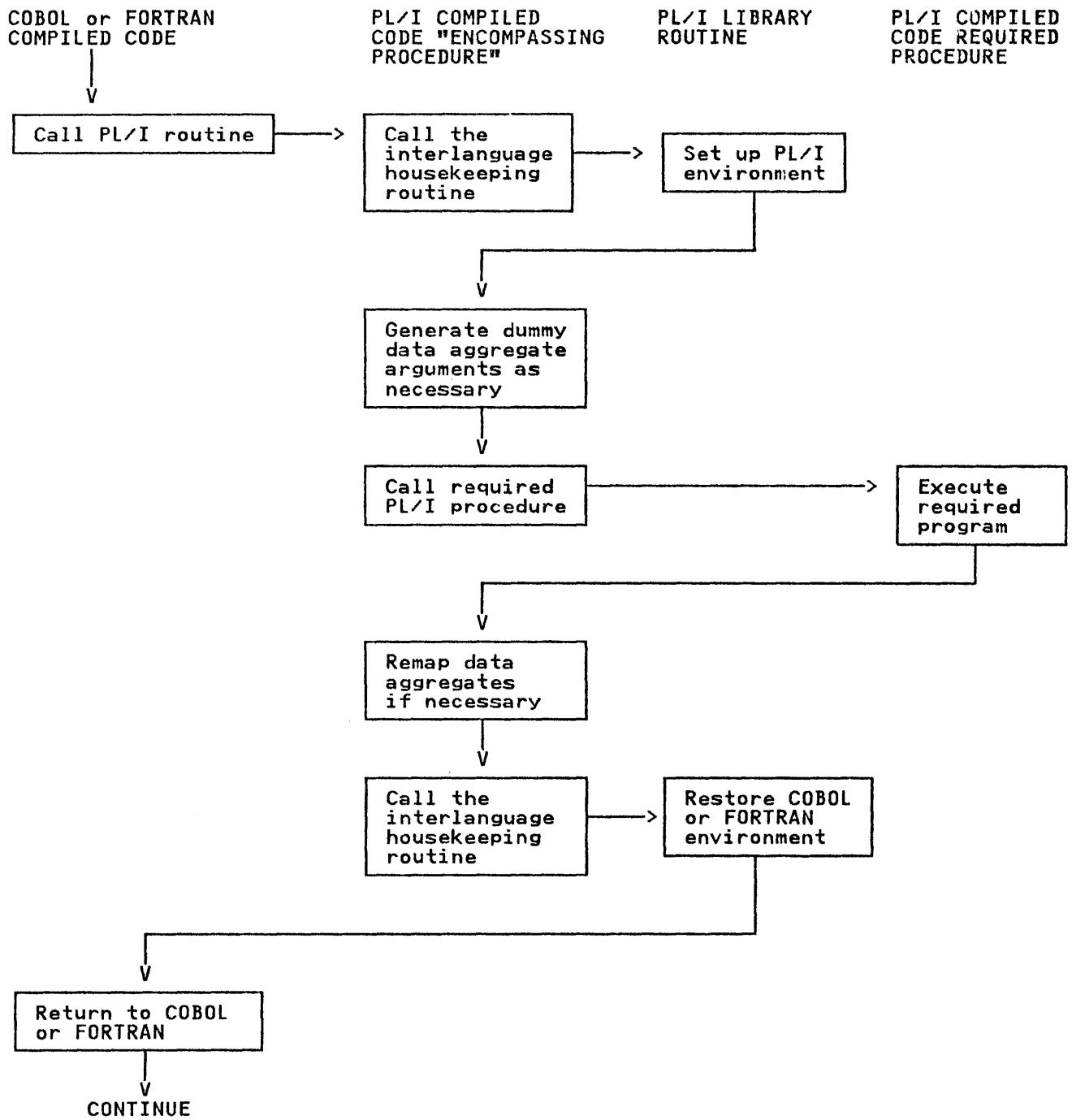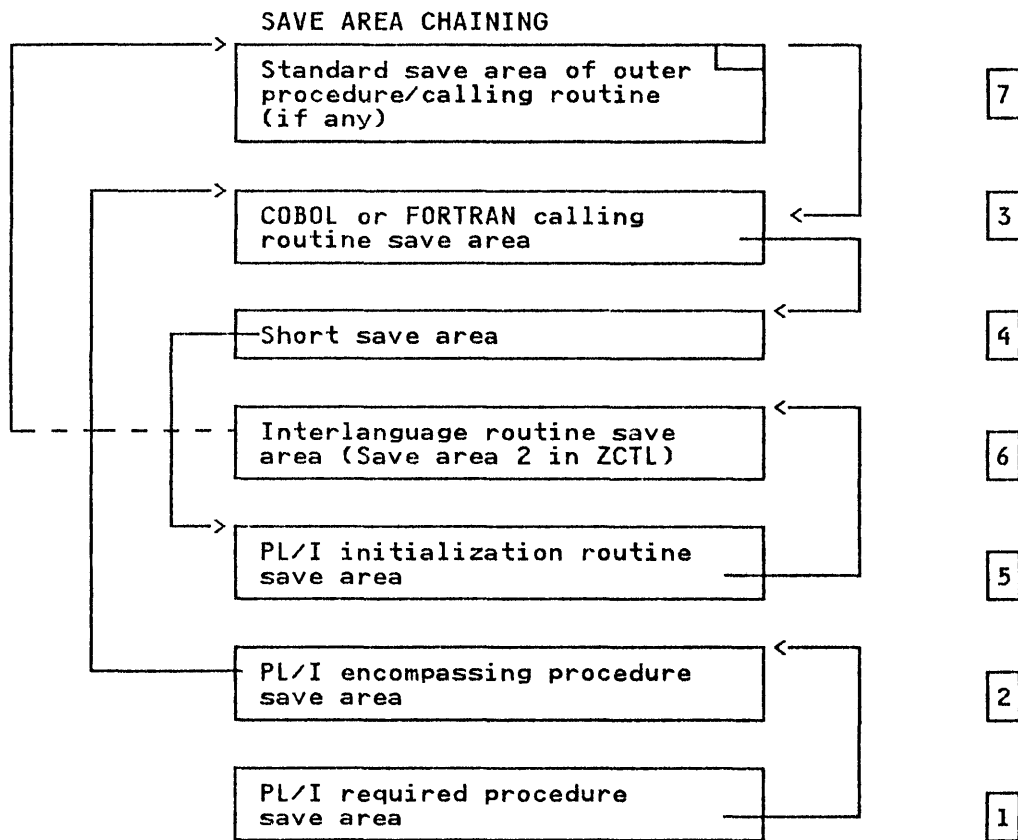
COBOL or FORTRAN
COMPILED CODE

PL/I COMPILED
CODE "ENCOMPASSING
PROCEDURE"

PL/I LIBRARY
ROUTINE

PL/I COMPILED
CODE REQUIRED
PROCEDURE

```
                   ┌─────────────────┐         ┌─────────────────┐        ┌─────────────────┐
                   │                 │         │ Call the        │        │                 │
                   │ Call PL/I       │ ─────>  │ interlanguage   │ ─────> │ Set up PL/I     │
                   │ routine         │         │ housekeeping    │        │ environment     │
                   │                 │         │ routine         │        │                 │
                   └─────────────────┘         └─────────────────┘        └─────────────────┘
```

```
                                               ┌─────────────────┐
                                               │ Generate dummy  │
                                               │ data aggregate  │
                                               │ arguments as    │
                                               │ necessary       │
                                               └─────────────────┘
```

```
                                               ┌─────────────────┐        ┌─────────────────┐
                                               │ Call required   │        │ Execute         │
                                               │ PL/I procedure  │ ─────> │ required        │
                                               │                 │        │ program         │
                                               └─────────────────┘        └─────────────────┘
```

```
                                               ┌─────────────────┐
                                               │ Remap data      │
                                               │ aggregates      │
                                               │ if necessary    │
                                               └─────────────────┘
```

```
                                               ┌─────────────────┐        ┌─────────────────┐
                                               │ Call the        │        │ Restore COBOL   │
                                               │ interlanguage   │ ─────> │ or FORTRAN      │
                                               │ housekeeping    │        │ environment     │
                                               │ routine         │        │                 │
                                               └─────────────────┘        └─────────────────┘
```

```
        ┌─────────────────┐
        │ Return to COBOL │
        │ or FORTRAN      │
        └─────────────────┘

           CONTINUE
```

Figure 113. The Sequence of Events When FORTRAN or COBOL Calls PL/I

SAVE AREA CHAINING

```
      >┌─────────────────────────────┐┐─────                    ┌───┐
     ┌┤ Standard save area of outer └┘     │                    │ 7 │
     ││ procedure/calling routine          │                    └───┘
     ││ (if any)                    │       │
     ││└─────────────────────────────┘       │
     ││   ┌─────────────────────────────┐   │              ┌───┐
     ││  >│ COBOL or FORTRAN calling    │<──┘              │ 3 │
     ││  ┌┤ routine save area           ├──┐               └───┘
     ││  ││└─────────────────────────────┘  │
     ││  ││  ┌─────────────────────────────┐<──┐           ┌───┐
     ││  ││ ┌┤─Short save area             │   │           │ 4 │
     ││  ││ ││└─────────────────────────────┘   │           └───┘
     ││  ││ ││ ┌─────────────────────────────┐<─┐            ┌───┐
     └┼──┼┼─┼┼─┤ Interlanguage routine save  │  │            │ 6 │
      │  ││ ││ │ area (Save area 2 in ZCTL)  ├─┐│            └───┘
      │  ││ ││ └─────────────────────────────┘ ││
      │  ││ ││>┌─────────────────────────────┐ ││          ┌───┐
      │  ││ └┼─┤ PL/I initialization routine │ ││          │ 5 │
      │  ││  │ │ save area                   ├─┘│          └───┘
      │  ││  │ └─────────────────────────────┘  │
      │  ││  │ ┌─────────────────────────────┐<─┘          ┌───┐
      │  └┼──┤ PL/I encompassing procedure │              │ 2 │
      │   │  │ save area                   ├──             └───┘
      │   │  └─────────────────────────────┘
      └───┘  ┌─────────────────────────────┐              ┌───┐
             │ PL/I required procedure     │              │ 1 │
             │ save area                   ├──            └───┘
             └─────────────────────────────┘
```

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 through 7 in the diagram show
the order of back-chaining).

Figure 114.   Chaining of Save Areas When PL/I is Called from a COBOL or FORTRAN
              Principal Procedure

HANDLING CHANGES OF ENVIRONMENT

Interlanguage Housekeeping Routines and their Control Blocks

Changes of environment are handled by three resident library
interlanguage housekeeping modules, one for calls to each
language. Common features are described below. A more detailed
description follows for each routine. The routines are:

    IBMBIEF for calls to FORTRAN
    IBMBIEC for calls to COBOL
    IBMBIEP for calls to PL/I

The job of these routines is the saving and restoring of
environments. This involves issuing SPIE/ESPIE macro
instructions suitable for the called routine and saving the PICA
or fake PICA of the calling routine so that a suitable

SPIE/ESPIE macro instruction is issued before returning. For PL/I, it also involves storing information about dynamic storage allocation and the TCA address.

The information required when setting up and storing environments is held in three chained control blocks:

1.   IBMBILC1: This is a control section that is link-edited from the resident library and included in each load module. It contains flags to indicate whether the PL/I, FORTRAN, or COBOL environment already exists and, if any do exist, contains a pointer to ZCTL.

2.   ZCTL: This holds PICA or fake PICA, TCA, and other error-handling addresses. It also holds flags indicating which languages are currently active.

     ZCTL is generated on the first of a series of interlanguage calls and is retained until that series of calls is completed. For calls to FORTRAN and PL/I, it is retained until the routine that made the first interlanguage call is itself terminated.

     Also held in ZCTL are the additional save areas used when the chaining is altered. These are known as save area 1, save area 2, and the ghost save area. The uses of these save areas are given in the individual module descriptions.

3.   ZVDAs (Interlanguage VDAs)

     These hold flags that indicate which languages were active before the latest call, the address of the caller's PICA or fake PICA, the address of the most recent PL/I DSA. ZVDAs are chained to the ZCTL, one for each interlanguage communication call.

An interlanguage VDA is acquired for every interlanguage call and discarded when the called routine is terminated. Interlanguage VDAs are held in the PL/I LIFO storage stack.

The methods of chaining used for these control blocks when PL/I is the called and the calling language is shown in Figure 115 on page 291 and Figure 116 on page 292. IBMBILC1 contains a pointer to ZCTL and ZCTL contains a pointer to the most recent interlanguage VDA. Interlanguage VDAs hold pointers to previous interlanguage VDAs, if any. If there are none, the pointer field is set to zero.

Figure 115. Example of Chaining Sequences (PL/I Principal Procedure)

❶ Initial situation

IBMBILC1 is set up as a control section by the PL/I interlanguage routines. Its first word and flags are initially zero.

IBMBILC1

Zero

Zero | Zero

COBOL   FORTRAN
flag.   flag

IBMBILC1   ZCTL

Address of ZCTL

X

Final situation ❽

ZCTL is retained until program is completed.

---

❷ 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/ESPIE.
5. Resets program check exit as required.
6. Returns to compiled code, which calls FORTRAN procedure.

IBMBILC1   ZCTL   VDA (First)

X

FORTRAN interrupt handling information

R13

R12

FORTRAN returns to PL/I (IBMBIEF) ❼

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/ESPIE macro.
3. Issues a PL/I STAE/ESTAE macro leaving the previously-stacked FORTRAN STAE/ESTAE for possible future use.
4. Returns control to compiled code.

---

❸ 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/ESPIE and STAE/ESTAE 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.

IBMBILC1   ZCTL   VDA (First)

X

FORTRAN interrupt handling information

R13

FORTRAN interrupt handling information

R12

PL/I returns to FORTRAN (IBMBIEP) ❻

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/ESPIE macro instruction to restore the error-handling situation to that of the calling routine. The information for the SPIE/ESPIE macro instruction is retrieved from the interlanguage VDA. The current PL/I STAE/ESTAE is canceled, leaving the previously stacked FORTRAN STAE/ESTAE in control.

The interlanguage routine returns control to the PL/I encompassing procedure, which then returns control to the FORTRAN program.

---

❹ 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/ESPIE 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.

IBMBILC1   ZCTL   VDA (Second)   VDA (First)

X | X

FORTRAN interrupt handling information

R13

R13

FORTRAN interrupt handling information

R12

COBOL returns to PL/I ❺

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/ESPIE macro instruction to restore the PL/I error-handling situation.

PL/I compiled code then continues.

Figure 116 (Part 1 of 2). Examples of Chaining Sequences (FORTRAN Principal Procedure)

```
┌─────────────────────────────────────────────────┐  ┌──────────────────────────┐
│                    IBMBILC1                      │  │                          │
│ FORTRAN calls PL/I    ┌─────┬───┬─────┐          │  │  ┌────────────────────┐  │
│                       │     │ X │     │          │  │  │ TCA etc.           │  │
│                       └─────┴───┴─────┘          │  │  ├────────────────────┤  │
│  ┌──────────>────────────>─────────────>────┐    │  │  │ Dummy DSA          │  │
│  │         │            │           │        │   │  │  ├────────────────────┤  │
│  │  ┌──────────┐   ┌──────────┐ ┌──────────┐ │   │  │  │ First VDA          │  │
│  │  │          │   │ R13      │ │ R13      │ │   │  │  ├────────────────────┤  │
│  │  ├──────────┤   ├──────────┤ ├──────────┤ │   │  │  │ Encompassing DSA 1 │  │
│  │  │ FORTRAN  │   │ FORTRAN  │ │ FORTRAN  │ │   │  │  ├────────────────────┤  │ ISA
│  │  │ interrupt│   │ interrupt│ │ interrupt│ │   │  │  │ Required DSA 1     │  │
│  │  │ handling │   │ handling │ │ handling │ │   │  │  ├────────────────────┤  │
│  │  │ inform.  │   │ inform.  │ │ inform.  │ │   │  │  │ Second VDA         │  │
│  │  ├──────────┤   ├──────────┤ ├──────────┤ │   │  │  ├────────────────────┤  │
│  │  │ R12      │   │          │ │          │ │   │  │  │ Encompassing DSA 2 │  │
│  │  └──────────┘   └──────────┘ └──────────┘ │   │  │  ├────────────────────┤  │
│ 4                                            │   │  │  │ Required DSA 2     │  │
│                                              │   │  │  ├────────────────────┤  │
│                                              │   │  │  │                    │  │
│                                              │   │  │  ├────────────────────┤  │
│                                              │   │  │  │ ZCTL               │  │
│                                              │   │  │  └────────────────────┘  │
└──────────────────────────────────────────────┘   └──────────────────────────┘
```

Figure 116 (Part 2 of 2). Examples of Chaining Sequences (FORTRAN Principal Procedure)

There is one interlanguage VDA for each call to another language. A VDA is set up when the call is made and discarded when the associated routine is terminated. The VDAs hold a record of the ZCTL flags that existed before they were called. These flags are placed in the VDA before the flags are altered and restored in ZCTL when the VDA is discarded. Thus, ZCTL always contains a record of the active language. This information is necessary when handling STOP statements.

The flags in IBMBILC1 contain a record of the environments that are active. These flags are used to test whether it is necessary to call the FORTRAN or PL/I initialization routines, or whether the environment can be restored from the information saved in ZCTL and the interlanguage VDAs.

Handling FORTRAN and PL/I Initialization/Termination Routines

FORTRAN and PL/I environments are set up by initialization routines and discarded by termination routines. To save the overheads of executing these routines on each call to the language, the save area for the termination routine is placed above that of the calling program. On the first call, the PICA or fake PICA address and, for PL/I only, the current DSA and TCA address are saved. For subsequent calls, this information is restored by the interlanguage routines and no call made to the initialization routine. Figure 117 on page 294 shows the principles involved.

```
+------------------+
| Save area for    | <---- ¬
| CALLER'S CALLER  |       |
+------------------+       |
         |                 |
         V                 |             +-------------------------------+
+------------------+       |             | Save areas are re-chained     |
| Save area for    |       |             | so that the environment       |
| CALLER           |       |             | is not discarded until        |
+------------------+       |        <----| the caller is terminated.     |
         |                 |             | Note: additional save         |
         V                 |             | areas are introduced.         |
+------------------+       |             | See Figure 118 on page 295    |
| Save area for    |       |             | and Figure 119 on page 296    |
| INITIALIZATION/  |       |             | for details.                  |
| TERMINATION      | - - --┘             +-------------------------------+
| ROUTINE PL/I or  |
| FORTRAN          |
+------------------+
         |
         V
+------------------+
| Save area for    |
| PL/I or FORTRAN  |
+------------------+
```
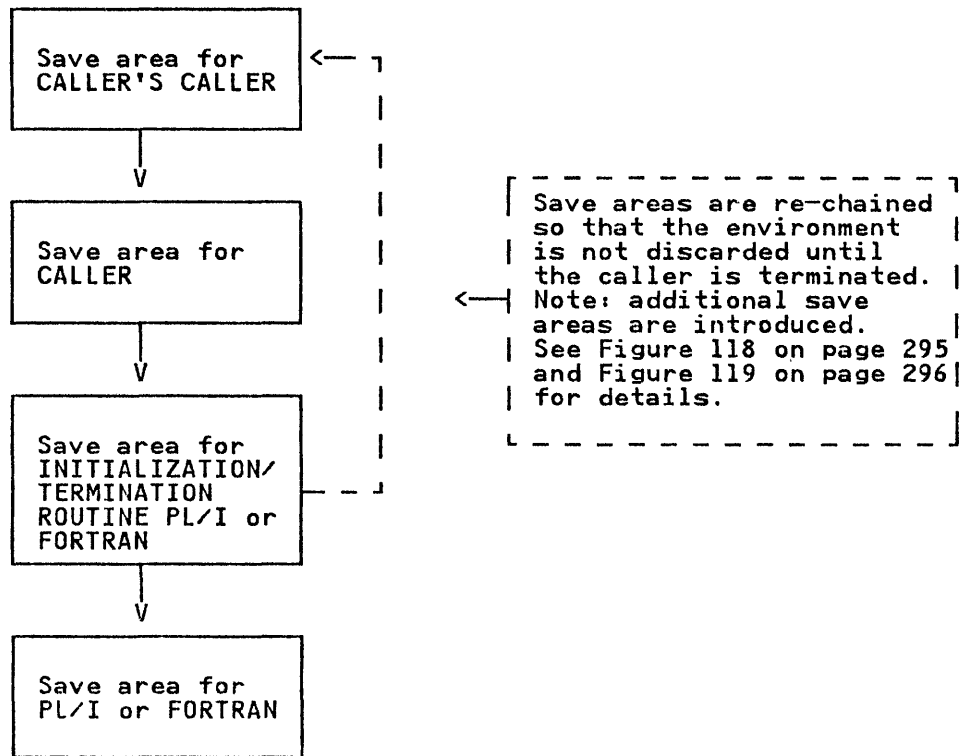
Figure 117. The Concept of Save Area Rechaining

The rearrangement of the save area chain results in certain
problems, for example, returning parameters from the caller to
the caller's caller. To overcome these problems, additional
save areas are inserted in the chain. These save areas result
in control passing to subroutines in the interlanguage
housekeeping routines known as tail code. Details are given in
Figure 118 on page 295 and Figure 119 on page 296 and in the
individual module descriptions that follow.

```
┌─────────────────────────────────────────────────────────┐
│                                                         │
│                  CALLER's caller                        │
│                                                         │
└─────────────────────────────────────────────────────────┘
                           A
                           │
┌─────────────────────────────────────────────────────────┐
│                    Save area 1                          │
│   (Register 14 in this save area points to              │
│   the tail code in IBMBIEF.  For normal                 │
│   return, the tail code calls IBCOM in                  │
│   the FORTRAN library to discard the                    │
│   FORTRAN environment.)                                 │
│                                                         │
└─────────────────────────────────────────────────────────┘
                           A
                           │
┌─────────────────────────────────────────────────────────┐
│                                                         │
│                      CALLER's                           │
│                                                         │
└─────────────────────────────────────────────────────────┘
                           A
                           │
┌─────────────────────────────────────────────────────────┐
│                                                         │
│                   FORTRAN routine                       │
│                                                         │
└─────────────────────────────────────────────────────────┘
```
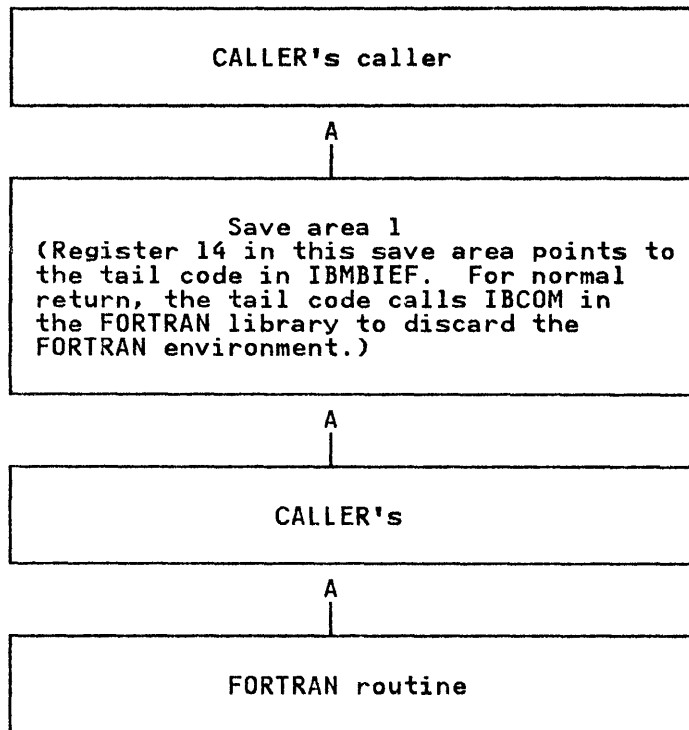
Figure 118.   Rechaining of Save Areas When FORTRAN Is Called
              from PL/I and the FORTRAN Environment Needs
              Initializing

Handling the INTER Option

When you specify the INTER option, you do not get the normal
PL/I interrupt handling, nor the normal interrupt handling for
the other language. Instead, you get PL/I error handling of
those interrupts that are left to the system by the non-PL/I
languages. To allow for this, the type of interrupt is analyzed
after it occurs and passed to the correct error-handling
routines.

Interrupts are analyzed by subroutines of the interlanguage
housekeeping routines known as traps.

During the normal call sequence, the interlanguage housekeeping
routines save the PICA or ESPIE TEST data for the called
language. The housekeeping routines then issue a SPIE/ESPIE
with options that return control to the trap code if an
interrupt occurs.

When an interrupt occurs, control is passed to the trap code to
analyze the interrupt. If the interrupt is normally handled by
the called language, control passes to that language's interrupt
processor. Otherwise, the interrupt is passed to the PL/I
error-handler for normal processing. The error-handler checks
for, and schedules any ON-units for that condition.

When the error-handler gains control, the trap routine saves the
interrupt information needed to restart the called program. The
trap routine also issues a special SPIE/ESPIE when the trap
routine is reentered because an ON-unit terminates normally.
This SPIE/ESPIE exit is used to restart the called program at
its next sequential instruction by forcing another program
check. The SPIE/ESPIE exit uses the operating system's
error-handler to restart at the point of the original interrupt.

```
    ┌ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ┐
           Caller's Caller
    | (replaced in this position |
      by tail code in IBMBIEP
    | R14 value restored from    |
      ghost save area)
    └ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ┘
```

```
┌─────────────────────────────┐
│         Save area 2         │
│   (passes control to tail   │
│      code in IBMBIEP)       │
└─────────────────────────────┘
               A
               |
┌─────────────────────────────┐
│         Dummy DSA           │
│    (Save area for PL/I      │
│       initialization        │
│    termination routine)     │
└─────────────────────────────┘
               A
               |
```

```
┌──────────────────────────┐         ┌──────────────────────────┐
│      Save area 1         │         │  Caller's Caller adapted │
│  Passes control to tail  │   <──── │  (register 14 value is   │
│  code in IBMBIEP (entered│         │  altered to pass control │
│  after FORTRAN STOP)     │         │  to tail code in IBMBIEP)│
└──────────────────────────┘         └──────────────────────────┘
                                                    A
                                                    |
         ┌─────────────────────────────┐
         │           CALLER            │
         └─────────────────────────────┘
                       A
                       |
         ┌─────────────────────────────┐
         │           PL/I              │
         │   Encompassing Procedure    │
         │           DSA               │
         └─────────────────────────────┘
                       A
                       |
         ┌─────────────────────────────┐
         │           PL/I              │
         │    Required Procedure DSA   │
         └─────────────────────────────┘
```

Figure 119.   Rechaining of Save Areas When PL/I Is Called from
FORTRAN or COBOL and the Environment Requires
Initialization

STOP and STOP RUN Statements

PL/I and FORTRAN STOP statements and COBOL STOP RUN statement cause certain problems because various save areas may be bypassed. The methods adopted to solve these problems are discussed in the individual descriptions of the modules.

HOUSEKEEPING MODULE DESCRIPTIONS

As the differences between individual interlanguage housekeeping modules are considerable, a detailed description of each module follows. The description covers the following situations:

1. When the associated language routine is called

2. When the associated language routine returns control

3. When an interrupt occurs with the INTER option

4. When a STOP or STOP RUN statement is executed

5. For PL/I and FORTRAN only, when the environment is discarded and the termination routine entered

COBOL WHEN CALLED FROM PL/I (IBMBIEC)

Before Entry to COBOL Program

IBMBIECA   Entry point for COBOL error-handling

IBMBIECB   Entry point for INTER error-handling

When IBMBIECA is called before the COBOL program, the following must be done:

1. Test to see if this is the first interlanguage call; if so, set COBOL flag in IBMBILC1 and set up ZCTL.

2. Acquire the interlanguage VDA and store register 12 and register 13 in the VDA. Write null PICA or fake PICA information in ZCTL.

3. If the INTER option is not specified (that is, entry point IBMBIECA), issue SPIE/ESPIE macro instruction so that errors are handled by the supervisor. Return to compiled code.

4. If the INTER option is specified (entry point IBMBIECB), issue new SPIE/ESPIE macro instruction and return so that interrupts are passed to the trap code.

On Return from COBOL Program (IBMBIECC)

The following actions take place on return:

1. A SPIE/ESPIE macro instruction is executed, which results in the PL/I error-handling scheme being restored.

2. The first word of the interlanguage VDA and the VDA flags are moved into the first word of ZCTL, and the VDA is freed.

Action on Interrupt in COBOL with INTER

If the INTER option is not specified, all program checks will be handled by the supervisor in the usual manner.

If the INTER option is specified and the 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 IBMBIECA, a SPIE/ESPIE macro instruction is issued, which results in interrupts being passed to the address in the trap code.

2. When an interrupt occurs, registers 12 and 13 are restored, thus restoring the PL/I environment.

3. A DSA is acquired for IBMBIEC in LWS. The address of the interrupt, in the second word of the PSW, is saved in the DSA and replaced by the address of another entry address in the trap. For underflow interrupts, the four bytes preceding the point of interrupt are also copied and placed before the trap in case the error-handler needs to examine them. The trap 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 SPIE/ESPIE 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.

RETURN FROM INTERRUPT: If there is a GOTO out of a PL/I ON-unit, control passes to the abnormal GOTO subroutine; this is because flags indicating an abnormal GOTO situation are set up by the trap code. The abnormal GOTO subroutine analyzes these flags and passes control to IBMBIEC, which handles any necessary housekeeping problems.

If the return is normal, the PL/I error-handling routines return control to the address in the second word of the PSW. This word has been altered by the code in the trap, and further trap code in IBMBIECA is entered.

It is necessary to return to the point of interrupt in the COBOL program without changing any of the register values and this can only be done via the supervisor. A new SPIE/ESPIE is set to point to further trap code and interrupt forced. The program is now in an interrupted state, the original INTER SPIE/ESPIE is reissued, and the registers and PIE are restored. The original interrupt address is set in the PSW. Control is returned to the supervisor, which passes control to the address in the PSW, with the correct register values restored.

ZERODIVIDE ON-Units

When used with certain COBOL compilers, normal return from a ZERODIVIDE ON-unit results in a data exception. This is because a zero and add decimal (ZAP) instruction is executed after the divide on Computational-3 data. The ZAP instruction picks up an invalid field.

Handling STOP RUN Statements

ANS COBOL STOP RUN statements are handled by a COBOL routine that passes control to a specified address. When IBMBIEC is called before entry to a COBOL program, this address is set to the tail code in IBMBIEC. This tail code dechains all save areas or routines that were entered after the PL/I caller and then executes a PL/I STOP statement.

FORTRAN WHEN CALLED FROM PL/I (IBMBIEF)

When FORTRAN is called by PL/I, IBMBIEFA 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 the FORTRAN Program

IBMBIEFA   Entry point for FORTRAN error-handling

IBMBIEFB   Entry point for PL/I INTER error-handling

Before the call to FORTRAN, IBMBIEFA does the following:

1. Tests the flags 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. IBMBIEFA stores register 13 in the interlanguage VDA, thus saving the PL/I environment.

3. 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-handling data is stored in ZCTL. Save area one (SA1) is then inserted into the save area chain. The resulting save area chaining is shown in Figure 118 on page 295.

4. IBMBIEFA acquires an interlanguage VDA. It points to 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.)

5. If the INTER option is not specified, it issues a FORTRAN SPIE/ESPIE macro instruction from ZCTL, sets program mask to '2', and returns to compiled code.

6. If the INTER option is specified, a SPIE/ESPIE macro instruction is issued that results in control passing to the trap should an interrupt occur. The program mask is reset to 'E' in case it was changed by the FORTRAN initialization routine. It then returns to the compiled code.

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 have been used as a function, entry point IBMBIEFD is used. Otherwise, entry point IBMBIEFC is used. The module IBMBIEF does the following:

1. A SPIE/ESPIE 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. The VDA flags are inserted in ZCTL and the VDA is 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 from registers, in which they were placed by the FORTRAN routine, to the location expected by PL/I.

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.

If the INTER option is specified, when an interrupt occurs, the following takes place:

1. When control passes from the supervisor to the ILC trap code, the type of interrupt is stored in 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 register 13 from the latest interlanguage VDA, thus restoring the PL/I environment.

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 an entry address in the trap in IBMBIEF.

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 SPIE/ESPIE macro instruction is issued to restore the PL/I error-handling situation. A branch is then made to the PL/I error-handler.

RETURN FROM INTERRUPT: If there is a GOTO out of a PL/I ON-unit, control passes to the abnormal GOTO subroutine; this is because flags indicating an abnormal GOTO situation are set up by the trap code. The abnormal GOTO subroutine analyzes these flags and passes control to IBMBIEF, which handles any necessary housekeeping problems.

If the return is normal, the PL/I error-handling routines return control to the address in the second word of the PSW. This word has been altered by code in the trap, and further trap code in IBMBIEFA is entered.

It is necessary to return to the point of interrupt in the FORTRAN program without changing any of the register values and this can only be done via the supervisor. A new SPIE/ESPIE is set to point to further trap code and an interrupt forced. The program is now in an interrupted state, the original INTER SPIE/ESPIE is reissued, and the registers and PIE are restored. The original interrupt address is set in the PSW. Control is returned to the supervisor, which passes control to the address in the PSW, with the correct register values restored.

Termination of Caller

When the PL/I program that called FORTRAN terminates, control is passed to the address held in the register 14 save area, in save area 1. This address is the address of the tail code in IBMBIEF. If the return is normal, the tail code calls IBCOM in the FORTRAN library to discard the FORTRAN environment (only if a FORTRAN environment exists). It then frees ZCTL and returns control to the caller's caller.

STOP Statements

If control returns to the tail code because of a FORTRAN STOP statement, the tail code discards any save areas that may have been bypassed by the FORTRAN STOP statement, and finally, executes a PL/I STOP statement, which terminates the program.

PL/I CALLED FROM COBOL OR FORTRAN (IBMBIEP)

As with the other interlanguage communication routines, IBMBIEP is called immediately before and immediately after the program that is to be executed. However, the interlanguage housekeeping routine cannot be called directly 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 entry 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.

Although the names of both PL/I procedures are the same, the encompassing routine gets control when called from either COBOL or FORTRAN. This happens because no ESD records are generated for the real entry points of the required PL/I program. Code for a PL/I encompassing routine is shown in Figure 114 on page 289. Figure 113 on page 288 shows the calling sequence.

Before Entry to PL/I Program (IBMBIEP)

Before a call is made to the required PL/I program, IBMBIEP 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 set, it means that a previous interlanguage call was made, and, because the call must have been made either to or from PL/I, the PL/I environment must have been set up.

   If it is established that the PL/I environment exists, register 12 is restored from ZCTL. A SPIE/ESPIE macro instruction is issued so that program checks are handled by the PL/I error-handler. The address of the old PICA or fake PICA is restored in the interlanguage VDA. Control returns to the encompassing routine.

3. If neither the COBOL nor the FORTRAN flag is on, PL/I is being called for the first time by a procedure in a program whose principal procedure is COBOL or FORTRAN. The following action is taken:

   a. IBMBIEP issues a GETMAIN macro instruction and sets up ZCTL in the storage acquired.

   b. The PL/I initialization routine, IBMBPIR, is called. It sets up the PL/I environment and returns control to an address in IBMBIEP instead of PLIMAIN. IBMBIEP then stores the registers of IBMBPIR in the dummy DSA.

c. The chaining of save areas is then altered, so that the dummy DSA (the save area used by IBMBPIR) is above the calling program's standard save area. The result of this is that, when the encompassing routine is complete, return is made to the COBOL or FORTRAN calling routine rather than to IBMBPIR. Thus, the PL/I termination routine is not entered and the PL/I environment is retained until the COBOL or FORTRAN calling program is completed.

Two further save areas are also inserted into the chain. These result in control passing to tail code in IBMBIEP, which handles housekeeping problems.

The save area of the caller's caller is also altered so that the register 14 value also points at tail code in IBMBIEP. The true register 14 value is saved in ZCTL in storage known as the ghost save area. The resulting save area chain is shown in Figure 119 on page 296. Action taken when the calling routine is terminated is described below, under "Termination of PL/I Environment."

4. A DSA for the encompassing routine is acquired.

5. The address of the new DSA is placed in the register 0 slot of the dummy DSA.

6. Control is then returned to compiled code in the encompassing routine.

Action after the PL/I Program Is Completed

Entry point IBMBIEPC—normal

Entry point IBMBIEPD—return value expected

IBMBIEP 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 a returned value may be required, 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 restores the caller's SPIE/ESPIE environments by issuing a SPIE/ESPIE macro instruction. This instruction restores the calling routine's program check exit, the PICA or fake PICA of which was stored in the interlanguage VDA.

Interrupt Handling

When PL/I is called by either COBOL or FORTRAN, error handling is carried out in the normal PL/I manner. The SPIE/ESPIE macro instruction is issued by IBMBPII when the PL/I environment is first set up. For calls after the first, the SPIE/ESPIE macro instruction is issued by IBMBIEP.

Termination of PL/I Environment

The PL/I environment is discarded when the caller is terminated. In a normal situation, control is returned by the caller to the address held in the register 14 save area of the caller's caller. This address was altered during the initialization of the PL/I environment to point to tail code in IBMBIEP.

This code receives control and rearranges the save area chaining. It then returns to IBMBPIR, whose registers are in the dummy DSA. The PL/I program then terminates, and control returns to save area 2.

This again points to tail code in IBMBIEP. This tail code restores the correct register 14 value of the caller's caller from the ghost save area and returns to the caller's caller.

STOP and STOP RUN Statements

For a PL/I STOP statement, the action is carried out in a normal manner, and flags in save area 1 indicate that an abnormal GOTO situation exists. The situation is analyzed by the abnormal GOTO subroutine, and control passes to the tail code whose address is held in save area 1.

For a FORTRAN STOP statement when the calling program is FORTRAN, the situation depends on how many levels of FORTRAN precede PL/I. If the caller is the highest level of FORTRAN prior to PL/I, control will be passed to <u>save area 1</u> and tail code entered to carry out the necessary housekeeping. If there is more than one level of FORTRAN, control will pass to the highest active level of FORTRAN and the job will be terminated without carrying out PL/I program termination.

A COBOL STOP RUN statement is intercepted by IBMBIEC, which then executes a PL/I STOP statement.

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, alignment requirements are met differently from PL/I. For an overview to data mapping between PL/I and other programming languages, see the <u>OS PL/I Optimizing Compiler: Programmer's Guide</u>.

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 <u>OS and DOS PL/I Language Reference Manual</u>.

Take, for example, a structure consisting of a single character and a fullword fixed binary item. The fullword fixed 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 EINARY (31,0)
```

and would be held thus:

```
+---------------------------------------+
| B |             C                     |
+---------------------------------------+
```
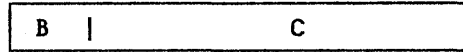
In COBOL, the structure would be declared:

```
01 A.
02  B,  PICTURE X, DISPLAY.
02  C,  PICTURE S9(9), COMPUTATIONAL.
```

and would be held thus:

```
+---------------------------------------+
| B |  3 unused bytes   |    C          |
+---------------------------------------+
```
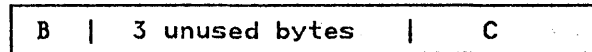
METHODS USED TO HANDLE DATA AGGREGATE ARGUMENTS

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 assigned to the
original argument when the execution of the called program is
completed.

If the data aggregates are not adjustable, 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 interlanguage
program.

The assignment of data between the dummy and the argument is
done by compiled code.

NOMAP, NOMAPIN, AND NOMAPOUT OPTIONS

You may use the NOMAP, NOMAPIN, and NOMAPOUT options to specify
that data aggregates are not to 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.

ASSEMBLER OPTION

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), parameter lists passed to the entry point are set up to contain no locator addresses. The addresses of any areas, arrays, strings, or structures are passed directly in a 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, "Communication between Routines" on page 64, 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 you require these facilities, you must either provide the necessary code yourself or use the COBOL option. 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 OS PL/I Optimizing Compiler: Programmer's Guide.

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, as described in "Handling Data Aggregate Arguments" on page 303. 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, no action is required. If the compiler cannot determine that the mapping will be the same, or if the structure is adjustable, both structures will be mapped. Adjustable structures will be mapped during execution by the resident library structure-mapping routines. Other structures will be mapped during compilation.

When reformatting of data is necessary, and when a record I/O statement involving a file with the COBOL option is executed, the following actions take place:

INPUT   The data is read into a structure that 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.

CHAPTER 14. MULTITASKING

INTRODUCTION

Multitasking allows the PL/I programmer to make use of operating system multiprogramming facilities within a single jobstep. The PL/I main procedure and certain other PL/I procedures are attached as tasks, and compete for the facilities of the CPU.

All features of the PL/I language that are implemented differently for multitasking and non-multitasking programs are handled by routines in the OS PL/I Resident and Transient Libraries. The non-multitasking resident library routines are held in the partitioned data set SYS1.PLIBASE; the multitasking resident library routines are held in the partitioned data set SYS1.PLITASK. When a multitasking program is link-edited, the automatic call library must be identified by sequential SYSLIB DD statements specifying first SYS1.PLITASK and then SYS1.PLIBASE.

Subroutines that have the same function in both the multitasking and the non-multitasking libraries have the same link-edit name). For further details, see "Naming Conventions" on page 54. Consequently, no special calls are required in compiled code. If the program uses multitasking, the multitasking version of the library module will be link edited, provided that SYS1.PLITASK is specified before SYS1.PLIBASE. Where a module is required only for multitasking programs, it is addressed from the TCA. The results of attempting to access such a module in a non-multitasking program are unpredictable. The concept of the multitasking library is shown in Figure 120.

```
┌─────────────────────────────────────────────┐
│                                             │
│         COMPILED CODE                       │
│                                             │
│   Relies on library modules for             │
│   executing code that is sensitive          │
│   to differences between multitask-         │
│   ing and non-multitasking.                 │
│                                             │
└─────────────────────────────────────────────┘
        │                        │
        V                        V
┌──────────────────────────┐  ┌────────────────────────────────┐
│                          │  │                                │
│ BASE LIBRARY SYS1.PLIBASE│  │ MULTITASKING LIBRARY SYS1.PLITASK│
│                          │  │                                │
│ Contains modules to handle│  │ Contains modules that handle all│
│ all library functions in a│  │ multitasking-sensitive operations.│
│ non-multitasking situation.│  │                                │
│                          │  │ These modules have the same link-│
│                          │  │ edit names as parallel modules in│
│                          │  │ the base library.              │
│                          │  │                                │
└──────────────────────────┘  └────────────────────────────────┘
```

Figure 120.   Multitasking Is Implemented by Use of a Multitasking Library

The use of a special multitasking library to handle all code that is affected by multitasking minimizes the effect on compiled code. Special action is required only for a CALL statement with any of the multitasking options, and for the epilog of a block that contains a CALL statement with multitasking options. Otherwise, the code generated for a

multitasking program is exactly the same as the code generated
for a non-multitasking program. The TASK option on a procedure
statement, necessary with some compilers, is ignored by the
optimizing compiler.

The Concept of the Control Task

To implement PL/I multitasking, the facilities offered by the
operating system control program have to be used in a manner
that meets the specifications of the PL/I language. Certain
facilities offered by PL/I, notably the ability of any task to
change the priority of any other task, are not directly
available in the system. Consequently, an interface is used
between the system facilities and PL/I tasks. This interface
takes the form of a <u>control task</u>.

The control task has all PL/I tasks attached as direct subtasks
and always has a higher priority than any PL/I task. Certain
functions are always carried out within the control task. These
functions are:

1. Attaching and detaching of tasks

2. Accessing or altering COMPLETION or PRIORITY values

3. Modification of event variables (except for STATUS
   pseudo-variable)

4. Generating PL/I dumps

5. Access to IOCBs in certain conditions

   For further details, see Chapter 8, "Record-Oriented
   Input/Output" on page 154.

The first two are carried out by the control task because of the
demands of the system control program. The third is carried out
by the control task because it is important that no two tasks
try to access the event variable chain at the same time.

The apparent and actual hierarchy of tasks is shown in
Figure 122 on page 309. The functions executed in the control
task are shown in Figure 121.

CONTROL TASK
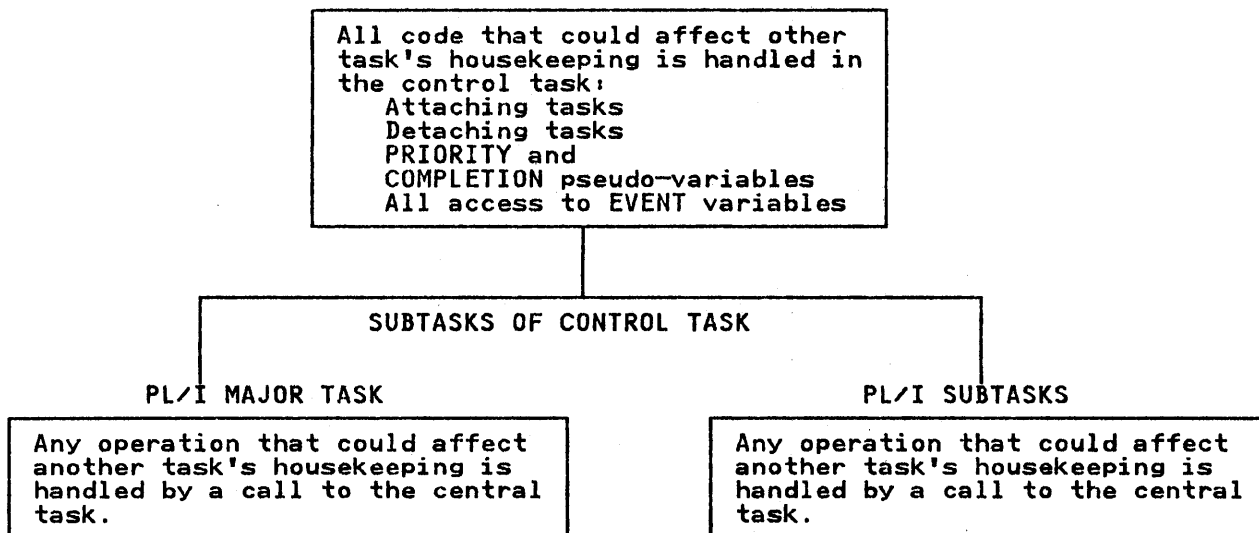
Figure 121. The Functions of the Control Task

```
PL/I PROGRAM

X:PROC;
•
•
•
CALL Y TASK (T1) EVENT (E2);
   Y:PROC;
   •
   •
   CALL Z TASK (T2) EVENT (E2);
    Z:PROC;
    •
    •
    END Z;
   END Y;
END X;
```

```
PL/I HIERARCHY                    ACTUAL HIERARCHY
                                  (as recognized by operating system)


                                         CONTROL TASK


                      MAJOR TASK



        Y (task T1)         MAJOR TASK      Y (task T1)      Z (Task T2)
        SUBTASK OF          SUBTASK OF      SUBTASK OF       SUBTASK OF
        MAJOR TASK          CONTROL TASK    CONTROL TASK     CONTROL TASK


Z(task T2)
SUBTASK OF Y
```
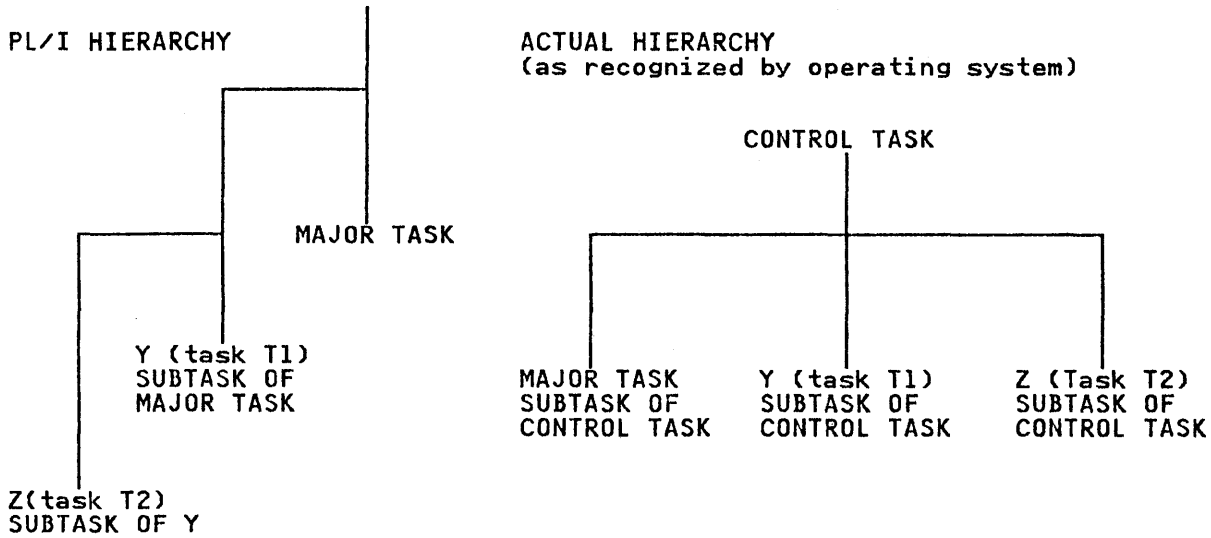
Figure 122. The Hierarchy of Tasks

Throughout most of the execution of a PL/I multitasking program,
the control task is in a wait state and the various PL/I tasks
are competing for the facilities of the CPU. The control task
waits on an ECB list that contains an ECB (event control block)
for each PL/I task and an ECB known as the task-end ECB that is
used when terminating a task. Whenever any of the functions
that must be carried out in the control task are required, the
ECB associated with the requesting task is posted with a request
code and the task goes into a wait sate, waiting on an ECB that
is posted complete when the requested function has been executed
in the control task.

Communication between Tasks

As explained above, there is no communication between PL/I tasks
except through the control task. Communication between the
control task and the PL/I tasks is made through control blocks
known as tasking appendages. Every PL/I task has a tasking
appendage, which is addressed from and is contiguous with the
TCA of the task.

As shown in Figure 123 on page 310, every tasking appendage is
headed by an ECB, followed by two fullwords for parameters,
followed by another ECB.

```
                          CONTROL TASKS ECB LIST
                         ┌────────────────────────┐
┌────────────────────────┤ ECB list element for   │   Control task waits on this
│                        │ major task             │   list until required to
│                        ├────────────────────────┤   perform a service
│                        │ ECB list element for   ├─────────────────────────────┐
│                        │ subtask                │                             │
│                        └────────────────────────┘                             │
│                                                                               │
│   PL/I MAJOR TASK                                    PL/I SUBTASK             │
│   TASKING APPENDAGE                                  SUBTASKING APPENDAGE     │
│ ┌──────────────────┐    ┌─Control tasks waits on ─┐  ┌──────────────────┐     │
└>│                  │    │ these ECBs which are    │  │                  │<────┘
  │   POST ECB       │ <──┤ posted with a code      ├─>│   POST ECB       │
  │                  │    │ indicating the service  │  │                  │
  │                  │    └─the subtask requires  ──┘  │                  │
  ├──────────────────┤                                 ├──────────────────┤
  │ PARAMETERS       │    ┌─                       ─┐  │ PARAMETERS       │
  │ Used by code     │ <──┤ Parameters further      ├─>│ Used by code     │
  │ executed in      │    │ define the service if   │  │ executed in      │
  │ control task     │    └─necessary             ──┘  │ control task     │
  ├──────────────────┤                                 ├──────────────────┤
  │                  │    ┌─PL/I task waits on this ─┐  │                  │
  │   WAIT ECB       │ <──┤ ECB. It is set complete  ├─>│   WAIT ECB       │
  │                  │    │ in the control task when │  │                  │
  │                  │    └─the required service    ─┘  │                  │
  │                  │       is completed              │                  │
  ├──────────────────┤                                 ├──────────────────┤
  │                  │                                 │                  │
  └──────────────────┘                                 └──────────────────┘
```

Figure 123.   The Post and Wait ECBs

The first ECB in the tasking appendage is known as the POST ECB,
and is one of the ECBs in the ECB list on which the control task
waits.   The second ECB is known as the WAIT ECB and is the ECB
on which the task waits while a function is carried out in the
control task.

When code within a subtask requires a service to be done in the
control task, it posts the POST ECB with a completion code to
identify the service required, and waits on its WAIT ECB.   The
WAIT ECB will be posted complete when the requested action has
been completed in the control task.

The completion codes that are used to post the POST ECB are:

X'0'   COMPLETION PSEUDO-VARIABLE POSTCODE

X'4'   EVENT ASSIGNMENT POSTCODE

X'8'   PRIORITY PSEUDO-VARIABLE POSTCODE

X'C'   I/O EVENT COMPLETION POSTCODE

X'10'   WAIT TERMINATION POSTCODE

X'14'   EXECUTE IN CTRL TASK

X'18'   DEDICATE CONTROL TASK ROUTINE

X'1C'   LIBERATE CONTROL TASK ROUTINE

X'20'   ATTACH A TASK

X'24'   END OF TASK

X'28'   TERMINATE SUBTASK

X'2C'   TERMINATE SUBTASK

Any parameters required are passed to the control task in the
list that follows the POST ECB.

Holding the Priority of the Task

The control program retains the priority of a task in an
associated TCB (task control block). At the PL/I level,
however, the priority is held in a <u>task variable</u>. This allows
the priority of the task to be set even when the task is
inactive, and also allows reference to the task by the program.
Each task has a task variable which is connected to the TCB
through the tasking appendage. The address of the associated
tasking appendage is placed in the task variable when the task
is attached.

When a change in the priority of a task is requested, the
priority is always changed in the task variable. If the task
variable is active, the priority is also changed in the TCB.

Also associated with a task is an event variable. The event
variable is set "complete" when the task is terminated.

All tasks have associated event and task variables. If none are
specified by the programmer, dummy variables are provided during
task attachment. These dummies are held in the task's own
workspace, and are discarded when the task is terminated.

<u>MULTITASKING HOUSEKEEPING</u>

Multitasking housekeeping is similar to non-multitasking
housekeeping. Every task has its own TCA and other blocks in
the program management area, as described in "The Program
Management Area" on page 81.

The major differences are that the TCA for each task has a
control block known as the tasking appendage, and that DSA
chaining between tasks cannot follow the rules of calling
procedures.

Control Name[1]	Link-edit Name[2]	Function
		Modules in the Tasking Library
IBMTEATA	IBMBEATA	Attention handling
IBMTJWTA	IBMBJWTA	WAIT statement
IBMTPIRA	IBMBPIRA	Program initialization and task housekeeping
IBMTTOCA	IBMBTOCA IBMBTOCB	COMPLETION pseudo-variable
IBMTTPRA	IBMBTPRA	PRIORITY pseudo-variable
		Multitasking Modules in the Transient Library
IBMTPJDA	IBMTPGDA	Storage management with REPORT
IBMTPJRA	IBMTPGRA	Storage management with NOREPORT
IBMTPIIA	IBMTPIIA	Program initialization (prior to Release 3.0)
IBMTPITA	IBMTPITA	Program termination
IBMTPJIA	IBMTPJIA	Program initialization
IBMTPJRA	IBMTPJRA	Program initialization and task housekeeping
IBMTTEPA	IBMTTEPA	ATTACH macro instruction entry point

Figure 124.   Modules in the Multitasking Library

Notes to Figure 124:

[1]   Control name is the name that uniquely defines the module.

[2]   Link-edit name is the name by which a module is known to the linkage editor. Multitasking and non-multitasking modules that handle similar functions have the same link-edit name.

As shown in Figure 125 on page 313, the chaining of DSAs is arranged so that the dummy DSA of the attached task is in the chain but the DSA of the attaching procedure is not. This protects the attached tasks from any changes in establishment of ON-units that may occur in the block that attached the task. In order that error handling and other functions using the back-chain may function correctly, certain items, such as ON-cells and dynamic ONCBs, are copied from the attaching task's DSA to the dummy DSA of the attached task at the time of attachment.

```
        MAJOR TASK TCA
      ┌──────────────────────┐
      │ Program              │
      │ Management Area      │
      ├──────────────────────┤              SUBTASK1 TCA                    SUBTASK2 TCA
  ┌─> │ MAJOR TASK           │         ┌──────────────────────┐        ┌──────────────────────┐
  │   │ DUMMY DSA            │         │ Program              │        │ Program              │
  │   ├──────────────────────┤         │ Management Area      │        │ Management Area      │
┌─┼─> │ MAIN PROC DSA        │ <────── ├──────────────────────┤ <───── ├──────────────────────┤
│ │   │                      │    ┌─>  │ DUMMY DSA            │    ┌─> │ DUMMY DSA            │
│ ┌─> ├──────────────────────┤    │    ├──────────────────────┤    │   ├──────────────────────┤
│ │   │ BEGIN BLOCK          │ <─┐│    │ Procedure            │ <─┐│   │ Procedure            │
│ │   │ DSA                  │   └┘─ ─ │ DSA                  │   └┘─ ─ │ DSA                  │
│ │   ├──────────────────────┤        ├──────────────────────┤        ├──────────────────────┤
└─┘   │                      │        │                      │        │                      │
      │                      │        │                      │        │                      │
      └──────────────────────┘        └──────────────────────┘        └──────────────────────┘
```
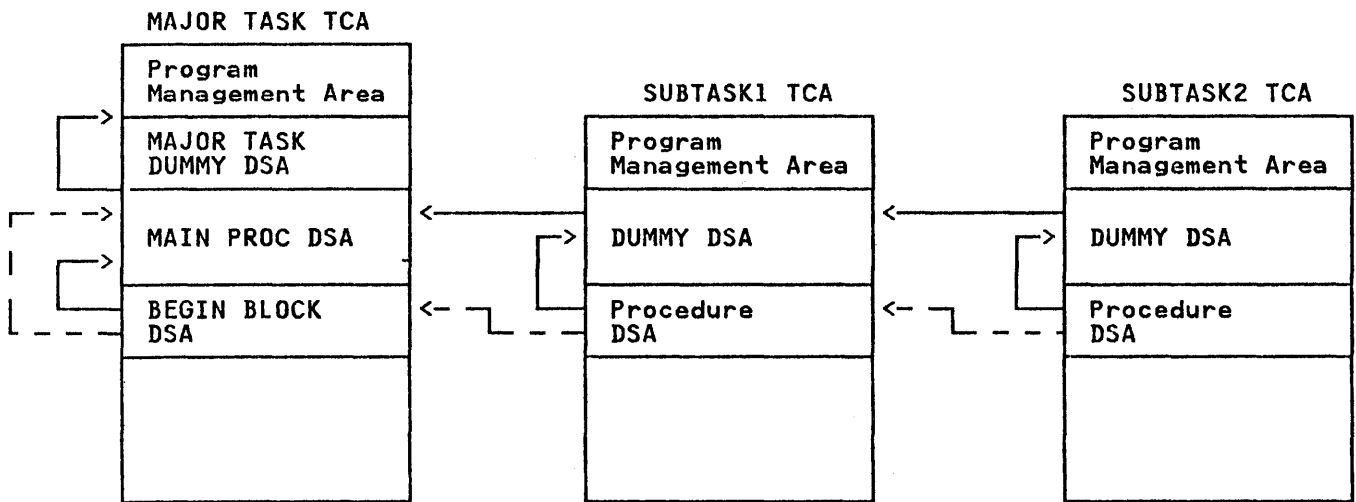
PL/I procedures involved

```
┌──────────────────Main procedure (major task)
│ ┌────────────────Begin block
│ │ ┌──────────────Procedure for subtask1
│ │ │ ┌────────────Procedure for subtask2
│ │ │ │
│ │ │ └───┐
│ │ │     │
│ │ └─────┘
│ │
│ └───────┐
│         │
└─────────┘
```
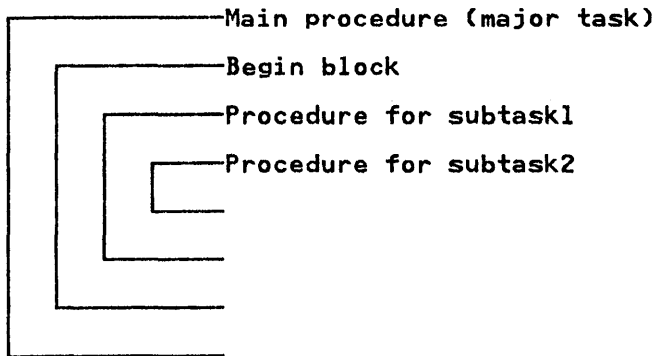
Note: To allow for inheritance of ON-units,
information held in the DSA of the attach-
ing task is copied into the dummy DSA of
the attached task.

Key
┌───┐
│ ─ ─ ─ ─ > Static back-chain   │
│   │
│ ──────────> Dynamic back-chain   │
└───┘
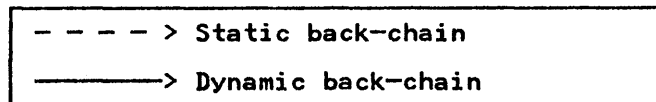
Figure 125.   Back-chains in Multitasking

If procedures executed as separate tasks are internal to one
another, a static back-chain is established through the DSAs.
This back-chain passes from the attached task's procedure DSA to
the DSA of the procedure in which the task was attached, and is
the same as for non-multitasking programs. This chaining allows
all internal procedures to access variables declared in outer
blocks without requiring special provision for multitasking.
(Special action is, however, necessary when handling the CHECK
condition.)

To maintain the PL/I hierarchy, more information than is
available in the DSA chain is required. In addition to the DSA
chain, tasks with the same attaching task are chained together,
and the most recently attached subtask is chained to its parent
task. The chains between tasks with the same attaching task are
known as __sibling task chains__. The sibling task chains and the

chain to the most recently attached subtask are all held within
the tasking appendage. The chaining arrangement, shown in
Figure 126, allows quick access to all related tasks.
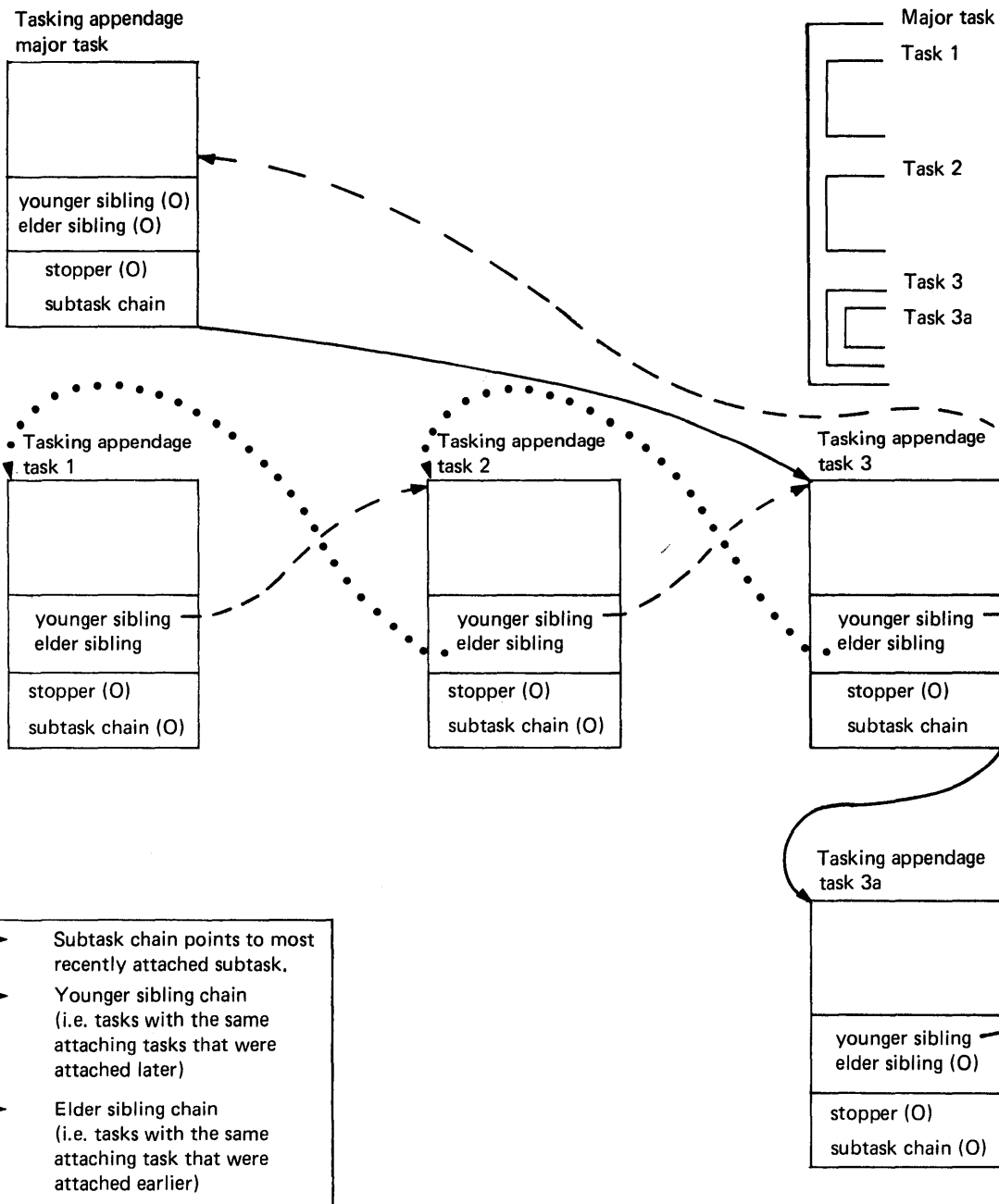
Note: Because tasks are chained in both directions, all relationships be quickly found.

Following the 'younger sibling chain' leads to the attaching task. When the attaching task is reached, the offset that should be the offset to the younger sibling is to the stopper. Thus it is known that the attaching task has been reached.

Figure 126. The Chaining of Tasks through Their Tasking Appendages

The sibling task chain goes in both directions. Each task is chained to the task attached immediately before it (elder sibling) and the task attached immediately after it (younger sibling). The most recently attached task has no younger sibling. Its younger sibling chain points instead to the attaching task. However, instead of pointing at the head of the tasking appendage, it points at offset X'8' within the tasking appendage. The effect of this is that an attempt to continue to follow the younger sibling chain results, beyond the attaching task, in access not to the younger sibling pointer but to a field offset from it by X'8'. This field, which is always set to zero in all tasks, is known as the stopper field. Access to it indicates that the attaching task has been reached.

When a task is terminated, all its subtasks must be terminated. To simplify finding these tasks, a flag is set in the DSA of the block in which a task is attached. The flag remains set while any active tasks are attached.

THE MULTITASKING LIBRARY

Module IBMTPIR loads IBMTPJR to perform most multitasking functions. IBMTPJR carries out the majority of functions that are executed in the control task. IBMTPJR issues a LOAD macro instruction to pass control to IBMTPJI to perform parameter translation, and to initialize the control task and the storage for the major task. IBMTPJR then attaches the major task. IBMTPJR also contains the instructions to handle the major functions which have to be carried out within the control task. Each of these functions is handled by a particular subroutine within IBMTPJR. A simplified flowchart of IBMTPJR is shown in Figure 127 on page 316.

The program initialization module IBMTPJR has a register save area, but is unlike other PL/I library routines in not having a DSA. IBMTPIR acquires workspace, contiguous with the standard register save area, to hold: the addresses of the ECB lists; the address of the area where the next ECB-list element will be placed; the task-end ECB (used when detaching a task—for further details see, "Detaching a Task" on page 318) the diagnostic file block, and the dump block. These last two blocks are held in the control task workspace because they must serve for all PL/I tasks.

Supporting IBMTPJR are two routines that are link-edited only when necessary: IBMTTOC is link-edited only if the COMPLETION pseudo-variable is used; IBMTPRA is link-edited only if the PRIORITY pseudo-variable is used.

Also included in the multitasking library are a number of routines that handle action which requires different machine instructions for a multitasking program. Among these routines are storage management and error handling routines.

All the routines in the multitasking library are shown in Figure 124 on page 312. The storage management routines and some of the tasking routines are described in, OS PL/I Transient Library: Program Logic, the remaining routines are described in OS PL/I Resident Library: Program Logic.

Figure 127. A Simplified Flowchart of IBMTPJR

Subroutines executed in control task

```
┌─────────────────────┐
│      ATTACH          │
├─────────────────────┤
│  Create a subtask    │
└─────────────────────┘

┌─────────────────────┐
│    COMPLETION        │
├─────────────────────┤
│ Set completion of    │
│ event variable       │
└─────────────────────┘

┌─────────────────────┐
│     PRIORITY         │
├─────────────────────┤
│ Set priority of      │
│ task variable        │
└─────────────────────┘

┌─────────────────────┐
│   EVENT ASSIGN       │
├─────────────────────┤
│ Set completion       │
│ and priority of      │
│ event variable       │
└─────────────────────┘

┌─────────────────────┐
│  COMPLETE EVENT      │
├─────────────────────┤
│ Used by              │
│ wait module          │
└─────────────────────┘

┌─────────────────────┐
│  DECHAIN EVTAB       │
├─────────────────────┤
│ Used by wait         │
│ module               │
└─────────────────────┘

┌─────────────────────┐     ┌─────────────────────┐
│ Execute in           │     │ Execute code         │
│ control task         │────▶│ indicated until      │
│                      │     │ free control call    │
│                      │     │ is reached           │
└─────────────────────┘     └─────────────────────┘

┌─────────────────────┐
│   END OF BLOCK       │
├─────────────────────┤
│ Detach tasks         │
│ if necessary         │
└─────────────────────┘

┌─────────────────────┐
│    END OF TASK       │
├─────────────────────┤
│ Terminate task       │
└─────────────────────┘

┌─────────────────────┐
│      STOP            │
├─────────────────────┤
│ Execute STOP         │
│ statements.          │
└─────────────────────┘
```

Main Program Flow (IBMTPJR)

```
         ( ENTRY )

┌─────────────────────┐
│ Initialize major     │
│ task                 │
└─────────────────────┘

┌─────────────────────┐
│ Initialization       │
│ common to            │
│ all tasks            │
└─────────────────────┘

┌─────────────────────┐
│ Post wait-ECB        │
│ complete             │
└─────────────────────┘

┌─────────────────────┐
│ Wait on              │
│ ECB list             │
└─────────────────────┘

      WAIT STATE

┌─────────────────────┐
│ Control resumed      │
│ here                 │
└─────────────────────┘

┌─────────────────────┐
│ Search ECB list      │
│ to determine which   │
│ service is required  │
│ and set ECB          │
│ incomplete           │
└─────────────────────┘

┌─────────────────────┐
│ Branch to            │
│ appropriate          │
│ subroutine           │
└─────────────────────┘

      ◇ Major task ◇ ── No
            │
           Yes
            │
         ( END )
```

Subroutines entered via TCA or TCA appendage from PL/I tasks

Call task
End task
Return
Get control

```
      ( ENTRY )

┌─────────────────────┐
│ Carry out            │
│ checking             │
│ set up               │
│ parameters           │
└─────────────────────┘

┌─────────────────────┐
│ Post post ECB        │
│ with correct         │
│ completion           │
│ code                 │
└─────────────────────┘

┌─────────────────────┐
│ Wait on              │
│ Wait ECB             │
└─────────────────────┘

      WAIT STATE

┌─────────────────────┐
│ Return to caller     │
└─────────────────────┘
```

Enqueue
SYSPRINT

```
      ( ENTRY )

┌─────────────────────┐
│ Check whether        │
│ SYSPRINT             │
│ enqueued             │
└─────────────────────┘

┌─────────────────────┐
│ Enqueue              │
│ SYSPRINT             │
│ if necessary         │
└─────────────────────┘

┌─────────────────────┐
│ Return to caller     │
└─────────────────────┘
```

Dequeue
SYSPRINT

```
      ( ENTRY )

  No  ◇ Enq'd at ◇
      ◇ this DSA ◇
      ◇ level?   ◇
            │
           Yes
            │
┌─────────────────────┐
│ Dequeue              │
│ SYSPRINT             │
└─────────────────────┘

┌─────────────────────┐
│ Return               │
└─────────────────────┘
```

Free control – executed in control task

```
      ( ENTRY )

┌─────────────────────┐
│ Restore DSA          │
│ of Get Control       │
│ routine              │
└─────────────────────┘

          ◯ A
```

HOW THE CONTROL TASK OPERATES

The control task is created by the system when the PL/I program is initialized. The instructions first executed within the control task are in the program initialization routine IBMTPIR. This routine is entered because its address is specified in the control section PLISTART. PLISTART is further described in "Link-Editing" on page 74.

IBMTPIR obtains a standard save area, and then loads and branches to IBMTPJR which performs the remainder of the initialization.

IBMTPIR sets up the environment for the major task, which it then attaches with an ATTACH macro instruction. After further initialization, control is given to the address held in PLIMAIN.

IBMTPJR then builds an ECB list which consists of the WAIT ECBs for the PL/I task that has been attached plus the task-end ECB. A wait is then issued on this ECB list, and the control task will remain in the wait state until the major task requires a service that must be handled in the control task.

When control returns to the control task, execution recommences in IBMTPJR immediately after the point at which the WAIT macro instruction was issued. The action at this point is to search the ECB list, discover which ECB has been posted, and then to carry out the action specified in the code posted in this ECB. The action is carried out by calling a subroutine of IBMTPJR. This subroutine may perform the function required, execute a sequence of requested instructions, or call further library routines to handle the requested function.

Whenever a new subtask is attached, a further POST ECB is added to the ECB list of the control task.

Whenever PL/I tasks require a service that is handled in the control task, a call is made to a library entry point. The majority of calls are to subroutines of IBMTPJR, which are addressed via the TCA or the TCA appendage. However, the PRIORITY and COMPLETION pseudo-variable routines are separate library modules. This saves space in programs where the pseudo-variables are not used.

ATTACHING A TASK

A CALL statement with one of the multitasking options is compiled as a call to an entry point in IBMTPJR. This entry point is addressed via a module list whose address is held in the TCA. The entry point is passed the address of the procedure that is to be executed as the attached task, and any parameters that are to be passed to that procedure.

The routine in IBMTPJR posts the POST ECB for the attaching task with a completion code of X'20', indicating that a new task is to be attached. It then issues a WAIT macro instruction on its own WAIT ECB, and the attaching task goes into the wait state.

Control passes to the control task. The first action of the code within the control task is to scan the ECB list to see which task is requesting a service, and which service is being requested. According to the completion code in the ECB, one of the subroutines in IBMTPJR is entered. For attaching a task, the attach-task subroutine is entered. The minimum storage the subroutine attempts to acquire is a new program management area. Depending on the options in the ISASIZE parameter, it may also attempt to acquire storage for DSAs and other dynamic requirements.

The new program management area is set up within the storage acquired, and the new TCA is placed at the head of the chain of child tasks that is held in the attaching task's TCA.

The new TCA is then associated with a task variable and an event variable. If these were specified in the CALL statement, they are used. Otherwise, dummy event and task variables are set up by IBMTPJR. These dummy variables are held in the working storage of the new block. The event and task variables are then chained to and from the TCA. A bit is set in the DSA of the block that was being executed when the task was attached.

The PRV of the attaching task is then copied into the attached task. This ensures that addressing information for files and controlled variables cannot be altered by the attaching task. Similarly, ON-unit establishment information is copied from the attaching task's current DSA into the dummy DSA of the attached task. This ensures that the subtask acts according to the situation prevailing at the time when the call was made.

The attaching routine finally sets the POST ECB of the new task incomplete, adds this new POST ECB to the control task's ECB list, completes the ECB on which the requesting task is waiting, and issues a WAIT macro instruction on the control task's ECB list.

The newly attached task and the original requesting task are now both ready to receive control from the control program. The control task is in a wait state, ready to service any further requests from PL/I tasks.

Failure of CALL...TASK Statements

A number of situations can cause a CALL...TASK statement to fail. These situations are:

1. Too many tasks are already active

2. There is insufficient storage for the new task

3. The task variable is already active

4. The event variable is already active

In any of these situations, the calling task is posted with a nonzero postcode. When this postcode is detected, the task generates the correct error code, and calls the error handler.

DETACHING A TASK

Tasks are normally detached when they reach any EXIT statement, or an END or RETURN statement in the procedure that was attached as a task. In such circumstances, control returns in the normal manner to IBMTPJR, whose registers have been stored in the dummy DSA of the task. IBMTPJR is then in a position to pass control to the control task, so that the requesting task can be terminated. After housekeeping operations, the control task sets the priority of the task to be detached as high as possible, completes the WAIT ECB of the task, and then waits on the task-end ECB. When the task to be terminated resumes control, it posts the task-end ECB complete, and terminates itself by returning to the control program.

The process described above is used because it is simpler than handling the ABEND that would otherwise result when one task is detached from another.

Abnormal Termination of a Task

When a block is terminated, any tasks attached during the execution of the block are also terminated. For this reason, epilog code of blocks in which tasks may be attached contains a call to a subroutine of IBMTPJR. This subroutine passes control to the control task, from which the purge task subroutine is called. This routine examines the DSA of the block being freed, to see whether any active subtasks remain; if any do remain, they are terminated.

Active subtasks are accessed via the chain of child tasks from the TCA of the task in which the block is being terminated.

Abnormal termination of a task involves ensuring that any WAIT statements being executed by the task are properly terminated, event variables are completed, task variables are set inactive, and ECB elements are removed. Event I/O operations started in the tasks are completed.

THE GET-CONTROL AND FREE-CONTROL ROUTINES

In order to increase the scope of jobs that can be handled within the control task, the program initialization routine includes a facility whereby a request can be made for any defined sequence of instructions to be executed within the control task. This facility is used by a number of library routines when accessing event variables, or carrying out other actions that have to be executed within the control task. It is not used by compiled code.

The instructions to be executed within the control task are delimited by calls to two library subroutines, whose addresses are held in the TCA. These routines are the get-control and free-control routines. Both are subroutines of IBMTPJR.

When the get-control routine is called within a PL/I task, it saves the caller's registers, posts its POST ECB, and issues a wait on the requesting task's ECB.

When the control task gains control, it restores the registers saved by the get-control routine, and branches to the address in register 14. The address will be the instruction after the call to the get-control routine, because the routine was called in the standard manner, that is, a BALR instruction on registers 14 and 15.

Execution of the instructions then continues in the control task until a call to the free-control routine is met. This routine stores the current registers in the DSA of the block that originally called the get-control routine. The free-control routine now posts the WAIT ECB of the requesting task, and resets the control task waiting on its ECB list.

During execution of the free-control routine, the routine modifies the value in the register 14 save area in the DSA of the block that originally called the get-control routine. When control returns to the original requesting task, it returns to the point in the get-control routine immediately following the point where the WAIT was issued. The get-control routine restores the register values, and branches to the new address in register 14.

The required instructions have now been executed within the control task, and execution can continue in the original task. The processes involved in the get-control and free-control routines can be followed in the flowchart of IBMTPJR in Figure 127 on page 316.

ALTERING COMPLETION AND PRIORITY VALUES

To prevent two PL/I tasks attempting to alter the completion and priority values of tasks or events at the same time, alteration of these values is always done by code in the control task.

When such access is required, compiled code in the requesting task branches to a library subroutine that posts the control task with a completion code in the POST ECB, and issues a wait in the requesting task. When the control task receives control, it inspects the completion code, and calls a subroutine in IBMTPJR. For the PRIORITY pseudo-variable, the subroutine in IBMTPJR calls a subroutine in IBMTTPR to handle the actual alteration. This is to save space in programs where the PRIORITY pseudo-variable is not used.

The subroutine accesses and alters the values as requested. Where necessary, a CHAP macro instruction is issued to alter the priority of a task.

EXECUTING THE WAIT STATEMENT

The WAIT statement can be used in both multitasking and non-multitasking programs. A description of WAIT in the non-multitasking situation is given in Chapter 11.

At the PL/I level, each WAIT statement is associated with one or more events, and each event is associated with an event variable. When the specified number of these event variables is set "complete," the wait is terminated.

PL/I event variables are not accessed by system wait macro instructions; they contain a pointer to the event's ECB. This ECB will have been nominated in the WAIT macro instruction issued to the system, and will be set complete when the associated event is complete. When the event is complete, the PL/I program can inspect the ECB, and complete the event variable.

The PL/I event variable cannot be used to indicate to all WAIT statements nominating the associated event that the event is complete. This is because an event variable may be associated with a further event immediately after completion of the event with which it was formerly associated. If more than one task is waiting, this may be before all the WAIT statements nominating the event are satisfied. See Figure 128 on page 321.

```
              TASK 1        TASK 2



              _____      _____

              WAIT (E1);    WAIT (E1);

              _____      _____




    READ FILE (A)
    INTO (B)
    EVENT (E1);










              V             V
```

Task 1 reuses the event variable E1. If task 1 acquires
control before task 2 on completion of the original event
with which E1 is associated, then the event variable on
which task 2 is waiting will be associated with an event
other than that originally intended. A mechanism to
supplement the event variable is therefore needed: This
mechanism is known as the EVTAB chain. In the above
example, the EVTAB chain would allow task 2 to determine
that the original event was complete.

Figure 128.   Reusing Event Variables, and the Need for the EVTAB
               Chain

To overcome this problem, a control block known as an EVTAB is
used. An EVTAB is generated for every WAIT statement. For
every event nominated in the statement, an EVTAB element is
produced, containing the ECB for the event and a pointer to
other EVTAB elements associated with the event. Thus, when an
event is completed in one task, the chain from the event
variable is scanned and any ECBs associated with the event are
set complete.

A further control block is used in the implementation of the
WAIT statement. This is the wait information table (WIT). A
WIT contains a record of any WAIT statements that are being
executed in a particular task. This information is used when a
task is being terminated, because any active events must be
removed from the chain that associates event variables with
EVTABs. Were this not done, the chaining of EVTABs would be
destroyed because the EVTABs in the terminated task would be
lost.

The chaining of the control blocks described above is shown in
Figure 129.

Chains and Pointers used during execution of
WAIT statement

ISA for task 1

| Program management area |
| LIFO storage |
| DSA for WAIT module |
| WIT |
| EVTABs |
| ECB-list
Addresses of associated
ECBs |
| Major free area |

6

6

6

ISA for task 2

| Program management area |
| LIFO storage |
| DSA for WAIT module |
| WIT |
| EVTABs |
| ECB-list
Addresses of associated
ECBs |
| DSA for ON-unit |
| DSA for WAIT module
called in ON-unit |
| WIT |
| EVTABs |
| ECB-list |
| Major free area |

5

4

2

2

1

3

9

1

3

8

IOCB

| Pointer to event
variable |
| ECB |
| |

7

EVENT VARIABLE

| PRIORITY and COMPLETION flags |
| Pointer to ECB in IOCB if I/O
event |
| Start of chain of EVTABs |
| Address of TCA appendage of
task requiring I/O (I/O event
only) |

3

1

Figure 129. Chains and pointers used in implementing the WAIT statement

Notes to Figure 129 on page 322:

1. EVTAB chain. Headed by the event variable. Connects all WAIT statements that use the same event variable, and enables the information that events are complete to be passed to all tasks.

2. WIT chain. Headed in the TCA. Connects all WAIT statements being executed in one task, and enables the EVTABs of these waits to be removed from the EVTAB chain when a task is terminated during a WAIT statement.

3. Event variable pointer. Held in EVTAB. Used to access event variables and search EVTAB chain.

4. ECBLIST element pointer. Held in EVTAB. Used to find associated ECB if event is an I/O event.

5. TCA appendage pointer. Held in EVTAB. Used during task termination.

6. EVTAB pointers. Held in WIT. Used to indicate number of EVTABs when declaiming during abnormal termination caused by GOTO out of block.

7. ECB pointer. Held in event variable. Used, for I/O events only, to identify associated event.

8. TCA appendage pointer. Held in event variable. Used, for I/O events only, during building of EVTABs to test whether I/O is active in the task.

9. ECB pointers. Held in ECB list. Used by supervisor to test whether events are complete.

The Wait Module IBMTJWT

The WAIT statement is executed by means of a call to the wait module, IBMTJWT. The module is passed a list of event variables and, optionally, a value indicating how many of the events must be completed before the wait is satisfied. If no value is specified, all events must be completed.

The wait module may be passed various types of event variables:

1. Active event variables. These are associated with:

   a. I/O or display events that were initiated in the current task.

   b. I/O or display events that were initiated in another task.

   c. Events associated with tasks.

2. Inactive event variables. These are associated with events that must be completed by use of the COMPLETION pseudo-variable.

3. Incompletable event variables. These are associated with events that have caused entry to an ON-unit because an I/O condition has been raised in the current task, and which cannot be completed because the ON-unit also specifies a wait on the event that is already being waited on.

If any of the events are incompletable, IBMTJWT checks to see whether the WAIT statement can be satisfied by completable events. If the WAIT statement cannot be satisfied, an attempt is made to complete all I/O and display events initiated in the current task, as other tasks may be waiting on these events. When these events are completed, and the associated ECBs in other tasks set complete, the error handler is called to terminate the current task.

If the WAIT statement can be satisfied by completable event(s), the incompletable event is ignored.

If any of the events are I/O or display events initiated in the current task, an ECB will already have been created for these events when the statement with the EVENT option was executed. This ECB must be accessed and waited on. Access is made through the event variable.

Note that for I/O events, a CHECK macro instruction is issued by the I/O transmitter. If all events are I/O events initiated in the current task, and all of them have to be completed, it is possible to use the CHECK macro instruction to satisfy the WAIT statement. The wait module passes the events one at a time to IBMBRIO. Return is made when the event is complete. The wait module then searches the EVTAB chain, setting any associated ECBs complete. It then passes the next event to IBMBRIO, continuing the process until all events are complete. If all events need not be completed, this method cannot be used, because one of the events nominated might prove incompletable and, consequently, the task would be terminated.

If the events are not I/O or display events initiated in the current task, the wait module builds an EVTAB element for the event, and associates it with the event variable. If only one event is involved, the wait module then issues a WAIT macro on the ECB; if more than one event is involved, the wait module places the address of the ECB in an ECB list on which a WAIT macro instruction will be issued.

If the wait module issues a WAIT macro instruction on an ECB list, control will return to the module when one or more of the ECBs has been completed.

The wait module scans the EVTAB elements and discovers which of the events has been completed. If the event is an I/O event in the current task, it will be necessary to complete the event variable and scan the EVTAB chain, completing ECBs in any tasks that are waiting on the event that has been completed. The ECBs are completed by calling a subroutine of IBMTPJR, which executes the necessary instructions in the control task. The subroutine completes the ECBs by means of a POST macro instruction.

If the wait is to be made on events that can only be completed in other tasks, the wait module issues a WAIT macro instruction specifying that all the events in the ECB list must be completed.

When all completed ECBs have been handled, the ECB list and the EVTAB elements are rebuilt for all events that are not complete. A further WAIT macro instruction is issued on the ECB list, and the process is continued until the necessary number of events have been completed.

If the number of events needed to satisfy the WAIT statement are complete, but further events remain incomplete, it is necessary to dechain EVTABs from the chains associated with the incomplete events. This is done by a call to a subroutine in IBMTPJR, which executes instructions in the control task to remove unneeded EVTAB elements from the EVTAB chain.

If the WAIT statement specifies only active events, no further action can be taken until the events are complete. Accordingly, the wait module issues a WAIT macro instruction specifying that all events have to be completed. Thus control will not return to the task until the wait is satisfied.

ENQUEUING AND DEQUEUING ON SYSPRINT

In order to protect error messages from interruption by other output to SYSPRINT, or from error messages in different tasks, the error message modules and all calls to SYSPRINT are enqueued and dequeued by means of a call to a subroutine in IBMTPJR, which issues the ENQ and DEQ macro instructions. A call is made immediately before and immediately after the output.

Similar action is taken on EXCLUSIVE files, for which the ENQ and DEQ macro instructions are issued by the library module IBMBPQD.

APPENDIX A. 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 OS 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 or AUTOMATIC storage.

How Addressed

From an offset from registers 3 or 13 known to compiled code

0	1	2	3	4
0	A(Area Variable)			
4	Length			
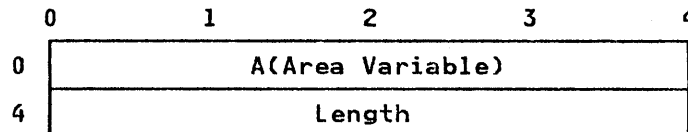
A(AREA VARIABLE): Is the address of the area variable control block.

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

At the head of the area variable.

```
        0           1           2           3           4
      ┌───────────┬─────────────────────────────────────┐
  0   │   Flag    │             Not Used                │
      ├───────────┴─────────────────────────────────────┤
  4   │       Offset of End Of Extent (OEE)             │
      ├─────────────────────────────────────────────────┤
  8   │     Offset of Largest Free Element (LFE)        │
      ├─────────────────────────────────────────────────┤
  C   │        End of Chain of Free Elements            │
      ├─────────────────────────────────────────────────┤
 10   │                                                 │
      │              Area Variable                      │
      │                                                 │
      └─────────────────────────────────────────────────┘
```

FREE ELEMENTS: 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'l' 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, "Communication between Routines" on page 64.

| When Generated

As far as possible during compilation. Adjustable values are filled in during execution.

| Where Held

Static internal control section or AUTOMATIC storage.

| How Addressed

From an offset from registers 3 or 13 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

```
        0                    1                  2
    0  |0      Offset to Entry for Containing    |
       |            Block                        |   Word 1
    2  |    Level          | Flag bits    DIM    |   Word 2
```
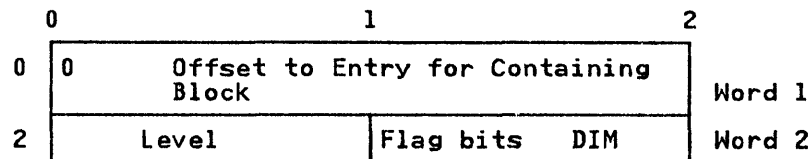
WORD 1

Bit 0 =   Always 0

Bit 1 = 1 Last element in the structure

The remaining bits comprise the offset to the entry for the containing block.

WORD 2: The first three bits of the DIM are flags.

FLAG BITS

Bit 0 = 1 Last element in structure

Bit 1 = 1 An AREA

Bit 2 = 1 BIT string

Bit 3 = 1 GRAPHIC string

Base Element

```
         0                    1                    2
    0  |1|Flag| Alignment   |      Length       |   Half-word 1
    2  |    Level    |Flag bits| DIM   |            Half-word 2
```
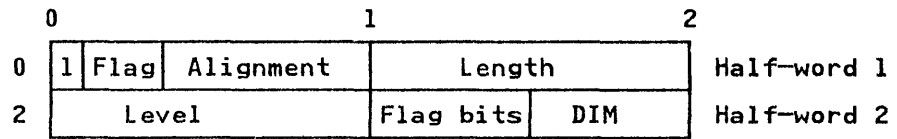
OFFSET: The offset within the aggregate descriptor descriptor to the entry for the containing structure. The offset is held in multiples of four bytes.

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=0 for strings and AREAs, whose length is held in descriptors.

FIRST ELEMENT MARKER: The first element in each structure has its offset field set to all '1'B.

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 or AUTOMATIC storage.

| How Addressed

| From an offset from registers 3 or 13 known to compiled code.

```
    0        1        2        3        4
  ┌────────────────────────────────────────┐
0 │        Address of data aggregate       │  Word 1
  ├────────────────────────────────────────┤
4 │          Address of descriptor         │  Word 2
  └────────────────────────────────────────┘
```
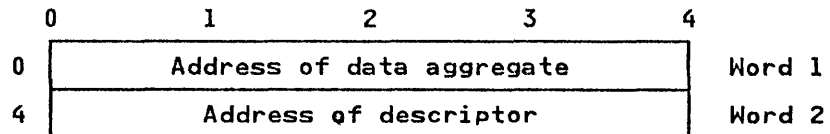
ARRAY DESCRIPTOR

Function

Contains information about the extent of an array. 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 or AUTOMATIC storage.

How Addressed

From an offset from registers 3 or 13 known to compiler code, or
from an aggregate locator.

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.

	0	1	2	3	4
0	RVO (AO—VO)				
4	Multiplier				
8	High bound		Low bound		
C	Multiplier 2				
10	High bound 2		Low bound 2		
	etc.				

Note: Two full words containing
multiplier and high and low bound are
included for each array dimension.

RVO: Relative virtual origin, the distance between the virtual origin (VO) and the 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 start of the first element in the array.

RVO is held as a bit value for arrays of unaligned bit strings, but otherwise as a byte value. Bit offsets are given in the string descriptor. Actual origin and virtual origin are also held as byte values.

HIGH BOUND: The highest subscript in any dimension.

LOW BOUND: The lowest subscript in any dimension.

MULTIPLIER: The multiplier is the offset between any two elements marked by the change of subscript number in any 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 bit string arrays, in which case they are bit values.

CICS APPENDAGE

Function

    Holds information needed during operation under CICS/OS/VS.

When Generated

    During program initialization under CICS/OS/VS.

Where Held

    In the program management area at the head of the ISA.

How Addressed

    From TCIC offset X'124' in TCA.

Offset	0 1 2 3 4	Label	Group
0	A(CICS TCA)	TCTCA	
4	A(CICS CSA)	TCCSA	
8	A(IBMFSTVA) or 0	TCSTV	
C	A(Msg Output Bootstrap)	TCTMS	
10	A(Report/Count Bootstrap)	TCTCR	
14	Terminal ID	TCTRM	
18	Transaction ID	TCTRN	
1C	PL/I Mask | CICS Mask | Command Workspace		
20	Temp1	TCTP1	
24	Temp2	TCTP2	>TCTMP
28	Temp3	TCTP3	
2C	A(DFHSAP, PL/I-CICS Nucleus Interface)	TCSAP	
30	A(PL/I to CICS Macro Interface)	TCMAC	
34	A(PL/I Program Exec. Interface Block)	TCEIB	
38	ABEND Code	TCABD	
3C	Interrupt Code	TCINT	>TCPSW
40	Return Address	TCRTN	
44	(PL/I Acquired Storage Chain)	TCSCH	
48	A(Buffer), Message/Count/Rep Records	TCBUF	
4C	Used as Exec. Interface DSA (184 bytes)	TCEIS	
104	User's Exec. Interface Block Copy (76 bytes)	TCEIC	

TCTMP: This area is used as a temporary workspace by PL/I. It
is comprised of the TCTP1, TCTP2, and TCTP3 fields.

TCPSW: This area holds the Program Status Word (PSW) at the
time of an interrupt. The field, TCINT, holds the interrupt
code; TCRTN holds the return address.

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

> From an offset in the PRV. (The PRV address is held at offset
> X'4' in the TCA.)

```
        0         1         2         3         4
      ┌─────────────────────────────────────────┐
   0  │                 PRVOFF                   │  Word 1
      ├─────────────────────────────────────────┤
   4  │                 Length                   │  Word 2
      ├─────────────────────────────────────────┤          Address
   8  │       Chain back to previous allocation  │  Word 3  Held in
      ├─────────────────────────────────────────┤          Pseudo-
   C  │           Task Invocation Count          │  Word 4  register
      ├─────────────────────────────────────────┤        <──────────
  10  │               Description                │  Word 5
      │       Field used for descriptor or       │
      │       locator/descriptor in certain      │
      │       circumstances, (see below)         │
      ├                                          ┤
      │                  Data                    │
      │                                          │
      │                                          │
      └─────────────────────────────────────────┘
```
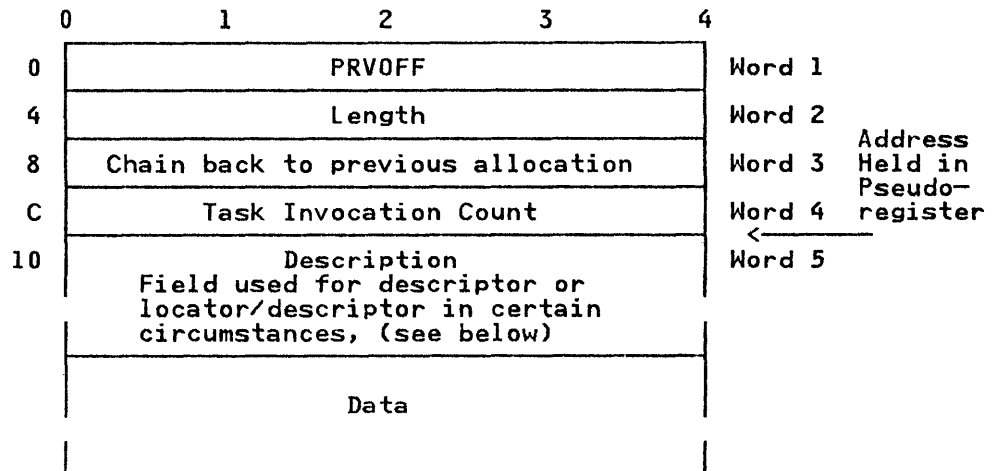
PRVOFF: Offset within pseudo-register vector associated with
the controlled variable.

LENGTH: Length of the total allocation including the 4 words of
the heading.

CHAIN BACK: Address of word 5 of previous allocation, set to
address of dummy FCB if first allocation

TASK INVOCATION COUNT: A method of identifying which task the
controlled variable is attached to. A controlled variable
cannot be freed within a task unless the task invocation count
of the variable is the same as that in the TCA.

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.

Thus for:

Strings and areas
> The controlled variable is headed by a
> locator/descriptor.

Structures and arrays
        The controlled variable is headed by a <u>locator</u>.

Structures and arrays with adjustable extents
        The controlled variable is headed by a <u>locator</u>
        followed by a <u>descriptor</u>.

All other data
        The <u>description field is not used</u> 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

From 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.

```
        0           1           2
  ┌─────────────┬───────────┬──────────────────┐
0 │   Flag 1    │  Flag 2   │ Further Bytes as │
  │             │           │ Required         │
  └─────────────┴───────────┴──────────────────┘
```
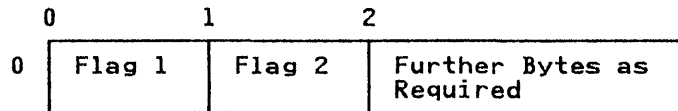
FLAG 1: Also known as Code Byte and Look up Byte, define the data type.

Hex Value	Data Type
X'00'	FIXED BINARY
X'04'	FIXED DECIMAL
X'08'	FLOAT
X'0C'	FREE DECIMAL (an internal form)
X'10'	FIXED PICTURE BINARY
X'14'	FIXED PICTURE DECIMAL
X'18'	FLOAT PICTURE BINARY
X'1C'	FLOAT PICTURE DECIMAL
X'20'	non-VARYING CHARACTER
X'24'	non-VARYING BIT
X'28'	VARYING CHARACTER
X'2C'	VARYING BIT

```
X'30'      CHARACTER PICTURE

X'40'      BINARY constant

X'44'      DECIMAL constant

X'48'      BIT constant

X'50'      F/E Format

X'54'      P Format (arithmetic)

X'58'      A/B/P Format (character)

X'5C'      C Format

X'60'      X Format

X'64'      COL Format

X'68'      SKIP Format

X'6C'      LINE Format

X'70'      PAGE Format

X'80'      LABEL

X'84'      ENTRY

X'88'      AREA

X'8C'      TASK

X'90'      OFFSET

X'94'      POINTER

X'98'      FILE

X'9C'      EVENT

X'A0'      GRAPHIC Fixed

X'A8'      GRAPHIC Varying
```

FLAG 2: completes the definition, if necessary.

```
Bits 0&1 =   00      A-format item
             01      B-format item
             10      Character picture format item
             11      GRAPHIC

Bit 2 =      0       Fixed constant
             1       Float constant

Bit 3 =      0       Not extended float
             1       Extended float

Bit 4 =      0       F-format/fixed picture
             1       E-format/float picture

Bit 5 =      0       Declared binary
             1       Declared decimal

Bits 4&5 =   11      Then DED is for character

Bit 6 =      0       Short precision
             1       Long precision
```

```
Bit 7 =        0        Real or length specified (A or B format)
                        aligned bit string.
               1        Complex (also set if E, F, or P in C-format) or
                        no length specified (A or B format) or unaligned
                        bit string.
```

All bits for which neither value is defined are set to '0'B

DED for STRING Data

```
      0              1              2
  0 | Flag 1 | Flag 2 |
```

DED for FLOAT Data

```
      0              1              2              3
  0 | Flag 1 | Flag 2 |precision|
```

DED for FIXED Data

```
      0              1              2              3              4
  0 | Flag 1 | Flag 2 |precision|scale±128|
```
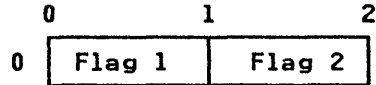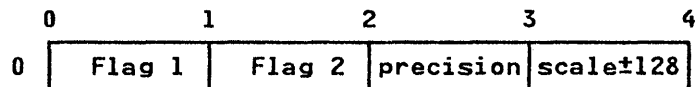
DED for PICTURE STRING Data

```
      0              1              2              3              4
  0 | Flag 1 | Flag 2 |         L1          |
  4 |        L2        | Picture in          |
                       | Internal Form       |
```
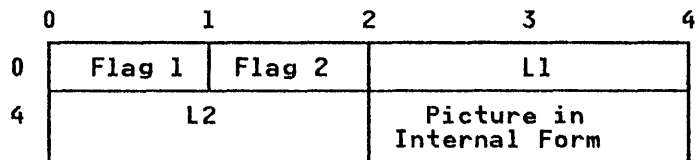
FLAG 1: (X'30')

L1: Length of field with insertion characters

L2: Length of field without insertion characters

INTERNAL CODE: The internal code for string pictures is as
follows:

Code   Picture(hex)

A   X'00'

```
        9       X'04'

        X       X'1C'
```

DED for PICTURE DECIMAL Arithmetic Data

```
        0           1           2           3           4
    ┌───────────┬───────────┬───────────┬───────────────┐
  0 │  Flag  1  │  Flag  2  │ Precision │    Scale      │
    │           │           │           │  Factor+128   │
    ├───────────┼───────────┼───────────┼───────────────┤
  4 │ Length of │ Length of │  Flag 3   │    Flag  4    │
    │  Picture  │   Data    │           │               │
    ├───────────┴───────────┴───────────┴───────────────┤
    │          Picture in internal code                 │
    └───────────────────────────────────────────────────┘
```

FLAG 1: (X'14' or X'1C')

FLAG 3: Describes the mantissa subfield.

Bit 0 =   Always set to zero

Bit 1 = 1   Drifting S in subfield

Bit 2 = 1   Drifting + in subfield

Bit 3 = 1   Drifting - in subfield

Bit 4 = 1   Drifting $ in subfield

Bit 5 = 1   Total suppression in subfield

Bit 6 = 1   ✳ in subfield

Bit 7 =   Always set to zero

FLAG 4: Describes the exponent subfield. It has the same
format as Flag Byte 3.

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
```

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, —, 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

```
    0           1           2
  ┌───────────┬───────────┐
0 │  Flag 1   │  Flag 2   │
  └───────────┴───────────┘
```
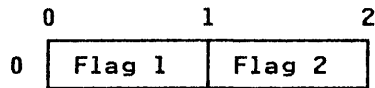
FLAG 1: (X'80, 84, 88, 8C, 90, 94, 98, or 9C')

FORMAT DEDS (FEDS)

For the meaning of the flag bytes, see "Data Element Descriptor (DED)" on page 337.

DED for F and E FORMAT Items (FED)

```
     0        1        2        3        4
   +--------+--------+------------------+
 0 | Flag 1 | Flag 2 |        W         |
   +--------+--------+------------------+
 4 |   D    |   X    |
   +--------+--------+
```

Flag byte 1 = X'50'

W Total length of the format field

D Number of decimal places

X Precision + 128 for F-format number of significant figures for E-format

DED for G FORMAT Items (FED)

```
     0        1        2        3        4
   +--------+--------+------------------+
 0 | Flag 1 | Flag 2 |      Length      |
   +--------+--------+------------------+
```

FLAGS

Flag 1 = X'A0' For G-format

Flag 2 = X'C0'

Length is optional.

DED for PICTURE FORMAT Arithmetic Items (FED)

```
     0        1        2        3        4
   +--------+--------+------------------+
 0 | Flag 1 | Flag 2 |        W         |
   +--------+--------+------------------+
 4 |  Copy of DED as for arithmetic picture |
   +----------------------------------------+
```

FLAG 1: (X'54')

W: Total length of the format field

DED for PICTURE FORMAT Character Items (FED)

```
         0         1         2         3         4
    ┌─────────────┬─────────────┬─────────────────────────┐
  0 │   Flag 1    │   Flag 2    │           W             │
    ├─────────────┴─────────────┴─────────────────────────┤
  4 │      Copy of DED as for arithmetic picture          │
    └─────────────────────────────────────────────────────┘
```

FLAG 1: (X'58')

W: Total length of the format field

DED for C FORMAT Items (FED)

```
         0         1         2         3         4
    ┌─────────────┬─────────────┬─────────────────────────┐
  0 │   Flag 1    │   Flag 2    │           W             │
    ├─────────────┴─────────────┼─────────────────────────┤
  4 │  FED for real part        │   FED for imag. part    │
    └───────────────────────────┴─────────────────────────┘
```

FLAG 1: (X'5C')

Note: The complex bit (bit 7) in Flag 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)

```
         0         1         2         3         4
    ┌─────────────┬─────────────┬─────────────────────────┐
  0 │   Flag 1    │   Flag 2    │       Parameter         │
    └─────────────┴─────────────┴─────────────────────────┘
```

FLAG 1: (X'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), is omitted for PAGE format.

DED for STRING FORMAT Items (FED)

```
         0         1         2         3         4
    ┌─────────────┬─────────────┬─────────────────────────┐
  0 │   Flag 1    │   Flag 2    │        Length           │
    └─────────────┴─────────────┴─────────────────────────┘
```

FLAG 1: (X'58')

The difference between A, B, and P (character) formats is given
by bits 0 and 1 of Flag 2. The length field may be omitted for
A and B format items.

DECLARE CONTROL BLOCK (DCLCB)

Function

Addresses file via PRV, holds declared file attributes, filename, and address of ENVB.

When Generated

During compilation.

Where Held

In a separate static control section for external files, or in a static internal control section for internal files.

How Addressed

The address is generated by the linkage editor for external files; It is addressed by an offset from register 3 for internal files.

	0	1	2	3	4	
0	Pseudo-Register Offset					DFCB
4	Declared Attributes					DCLA
8	Invalid OPEN Attributes					DOPA
C	A(Environment Block)					DENV
10	Offset of Graphics Extension		Offset of Filename Length			
14	Filename Length		Filename (to 31 characters)			

DECLARED AND INVALID OPEN ATTRIBUTES

Byte Number	Hex. Value	Attributes
1	01	STREAM
	02	RECORD
	04	DISPLAY
	10	reserved for (STRING)
2	01	SEQUENTIAL
	02	DIRECT
	04	TRANSIENT
	10	INPUT
	20	OUTPUT
	40	UPDATE
	80	BACKWARDS
3	01	BUFFERED
	02	UNBUFFERED
	04	KEYED
	08	EXCLUSIVE
	10	PRINT
4		Not used

DIAGNOSTIC FILE BLOCK (DFB)

Function

Holds information used by the error message routines.

When Generated

During program initialization.

Where Held

Program management area.

How Addressed

From X'40' in the TCA (TDFB).

	0	1	2	3	4	
0	Flags	Not Used	Rtn. Code	Not Used		
4	A(Transmitter)					ABTS
8	A(SYSPRINT DCLCB)					ASPD
C	A(Explicit Open)					AOCL
10	A(Improvised SYSPRINT DCB)					ASDC

FLAGS (AFLA)

AWTO	Bit 0 = 1	Messages going to operator's console
ASNO	Bit 1 = 1	SYSPRINT not open at the first attach.
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 prolog 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).

How Addressed

From register 13.

	0	1	2	3	4	
0	Flag 0	Flag 1	Not Used			
4	A(Chain Back)					CCHB
8						
C	Save Area R14					
10	Save Area R15					
14	Save Area R0					
18	Save Area R1					
1C	Save Area R2					
20	Save Area R3					
24	Save Area R4					->CRSA
28	Save Area R5					
2C	Save Area R6					
30	Save Area R7					
34	Save Area R8					
38	Save Area R9					
3C	Save Area R10					
40	Save Area R11					
44	Save Area R12					
48	A(LWS)					CLWS
4C	Segment #	A(NAB)				CNAB
50	Segment #	End of Prolog NAB				CEPN or CAPP
		or				
	A(TIA) or A(TTA) in Dummy DSA					
		or				
	To Save TFB1	Number of DSAs	Not Used			
54	Block-Enable Bits CENA		Current-Enable Bits CCNA			
58	A(Attaching DSA) in Dummy					CAAD
5C	A(First Static ONCB)					CSON
60	A(Most Recent Dynamic ONCB in Block)					CDON
64	Not Used					
68	Not Used					
6C	Reserved for the Checkout Compiler					
70	A(ONCELLS)					CAOC
74	Reserved Checkout Compiler	Imple- mentation Defined	Flags 2	Control Task Flag		

FLAGS

FLAG 0 (CFF0)

CDSA	Bit 0 = 1	LWS DSA
CONC	Bit 1 = 1	ON-cells exist
COCH	Bit 2 = 1	Dynamic ONCBs allocated
CIMP	Bit 3	Reserved for the Checkout Compiler
CBEG	Bits 4 & 5	00 Procedure DSA
		01 Begin DSA
		10 Library DSA
		11 ON-unit DSA
CDUM	Bit 6 = 1	Dummy DSA
CSUB	Bit 7 = 1	Subtask dummy DSA

FLAG 1 (CFF1)

CFCM	Bit 0 = 1	Byte CFFC is meaningful
CRNB	Bit 1 = 1	Restore NAB on GOTO
CRCE	Bit 2 = 1	Restore current-enable on GOTO
COVR	Bit 3 = 1	Callee can use this DSA
CGTO	Bit 4 = 1	EXIT DSA
CSNT	Bit 5 = 1	Statement number table exists
CSYE	Bit 6 = 1	SYSPRINT is enqueued by this block.
CFFB	Bit 7 = 1	Flags in Flags 2 are valid

FLAGS 2 (CFF2)

C2LD	Bit 0 = 1	Last PL/I DSA
C2ID	Bit 1 = 1	Ignore DSA for SNAP
C2IN	Bit 2 = 1	ILC DSA after interrupt
C2IC	Bit 3 = 1	Invocation Count in this DSA
C2SY	Bit 4 = 1	Symbolic dump for this DSA
C2FL	Bit 5 = 1	There are TSO line numbers
	Bits 6 & 7	Not used

CONTROL TASK FLAG

CCFC	Bit 0 = 1	Block has active subtasks
	Bits 1-7	Not used

This flag byte is the only one in the DSA used by the control task without synchronizing with the subtask. The subtask must never change it. This prevents interference between CPU's on a multiprocessing machine.

DUMP BLOCK (DUB)

Function

To hold information about the dump file.

When Generated

During program initialization.

Where Held

In the program management area.

How Addressed

From offset X'20' in the TIA (TDUB).

	0	1	2	3	4	
0	Flags 1	Flags 2	Not Used			
4	A(DCB)					ADCB
8	A(Buffer)					ABUF
C	A(Dump Transmitter)					ADXT
10	Current Line Number		Pagesize			
14	Not Used					
18	Not Used					
1C	A(PLIDUMP SYNAD Exit)					ASYN

FLAGS

FLAGS 1

ANDE	Bit 0 = 1	Dump file cannot be opened
	Bits 1-3	Not used
ANDH	Bit 4 = 1	PLIDUMP heading required
	Bits 5-7	Not used

FLAGS 2

ANSS	Bit 0 = 1	No subtasks' subpools
	Bits 1-7	Not used

ENTRY DATA CONTROL BLOCK

Function

Holds information that will enable an entry to be branched to and the correct static back-chain to be set up. Is used as an entry variable or when an entry is passed as a parameter.

When Generated

When the variable is allocated.

Where Held

Depends on the storage class of the data item.

How Addressed

Depends on the storage class of the data

```
     0          1          2          3          4
   ┌──────────────────────────────────────────┐
 0 │           Address of entry point          │  Word 1
   ├──────────────────────────────────────────┤
 4 │    Address of Statically Containing        │  Word 2
   │    DSA at Time of Assignment               │
   └──────────────────────────────────────────┘
```
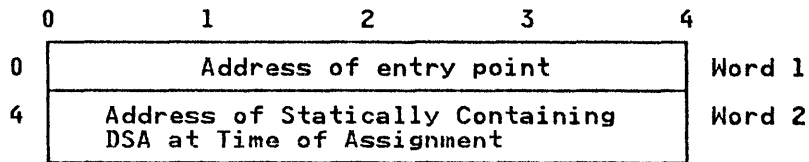
WORD 1

Bit 0 = 0   Address of entry
Bit 0 = 1   Address of location containing 8-char. EBCDIC name
             of entry point

WORD 2: Bit 0 is always set to zero.

ADDRESS OF STATICALLY CONTAINING DSA: This address is set in register 5 when the assignment is made to the variable. It enables variables in other blocks to be accessed. When assignment is made the address of the current statically containing DSA is set. This will be the correct address for the entry. If it were not, the entry itself would not be known.

ENVIRONMENT BLOCK (ENVB)

Function

> Holds environment options for a file so that the file may be
> correctly opened during execution.

When Generated

> During compilation

Where Held

> In a static control section with the DCLCB for external files.
> In static internal storage for internal files.

How Addressed

> From offset X'C' in the DCLCB

Offset	0	1	2	3	4	
0	NFLA	NFLB	NFLC	NFLD		
4	NFLE	NFLF	NFLG	NFLH		
8	Not Used					
C	A(Blocksize) or A(Pagesize 2260)					NBLK or NPAG
10	A(Record Length) or A(Linesize 2260)					NREC or NLIN
14	A(Number of Buffers)					NBUF
18	A(KEYLOC Value) or A(Attention Variable)					NLOC or NATN
1C	A(KEYLENGTH)					NKYL
20	A(BUFFOFF Value) or A(INDEXAREA Size)					NOFF or NNDX
24	A(NCP Value) or A(Size of ADDBUF)					NNCP or NADD
28	A(Password String Locator)					NPAS
2C	A(BUFND Value)					NBND
30	A(BUFNI Value)					NBNI
34	A(BUFSP Value)					NBSP

FLAGS

NFLA

NCON	Bit 0 = 1	Consecutive
NIND	Bit 1 = 1	Indexed
NRG1	Bit 2 = 1	Regional (1)
NRG2	Bit 3 = 1	Regional (2)
NRG3	Bit 4 = 1	Regional (3)
NTPM	Bit 5 = 1	TP(M)
NTPR	Bit 6 = 1	TP(R)
NOTH	Bit 7 = 1	Other organization

NFLB

NFIX	Bit 0 = 1	Fixed
NVAR	Bits 0 & 1	10 Variable
NUND		11 Undefined
NDEC	Bit 2 = 1	Decimal
NTRO	Bit 2 = 1	TRKOFL>
NBLO	Bit 3 = 1	Blocked
NSPA	Bit 4 = 1	Spanned
NASA	Bit 5 = 1	CTLASA
N360	Bit 6 = 1	CTL360
NEGS	Bit 7 = 1	GRAPHIC

NFLC

NLVE	Bit 0 = 1	LEAVE
NRRD	Bit 1 = 1	REREAD
NGKY	Bit 2 = 1	GENKEY
NCBL	Bit 3 = 1	COBOL
NOWR	Bit 4 = 1	NOWRITE
NXAR	Bit 5 = 1	INDEXAREA
NTOT	Bit 6 = 1	TOTAL
NXAS	Bit 7 = 1	INDEXAREA with no argument

NFLD

NBUU	Bit 0 = 1	BUFFERS
NCPF	Bit 1 = 1	NCP
NFPS	Bit 2 = 1	PASSWORD
NKEL	Bit 3 = 1	KEYLENGTH
NKLC	Bit 4 = 1	KEYLOC
NVFY	Bit 5 = 1	VERIFY
NNOL	Bit 6 = 1	NOLABEL
NABF	Bit 7 = 1	ADDBUF

NFLE

N226	Bit 0 = 1	2260
NLOK	Bit 1 = 1	Lock (2260)
	Bits 2-3	Not used
NSTL	Bit 4 = 1	SCALARVARYING
NUSA	Bit 5 = 1	ANSCII
NBOF	Bit 6 = 1	BUFOFF
NBFL	Bit 7 = 1	BUFOFF(L)

NFLF

NXML	Bit 0 = 1	Index multiple
NX11	Bit 1 = 1	High index 2311
NX14	Bit 2 = 1	High index 2314
NOTM	Bit 3 = 1	No tape mark
NALT	Bit 4 = 1	Alternate tape

```
NOFT    Bit 5 = 1       OFL tracks
NXTN    Bit 6 = 1       Extent number
        Bit 7           Not used
```

NFLG

```
NFFM    Bit 0 = 1       F-format
NVFM    Bit 1 = 1       V-format
NUFM    Bit 2 = 1       U-format
NSP2    Bit 3 = 1       Spanned
NBL2    Bit 4 = 1       Blocked
        Bits 5-7        Not used
```

NFLH

```
NVSM    Bit 0 = 1       VSAM
NFBD    Bit 1 = 1       BUFND
NFBI    Bit 2 = 1       BUFNI
NFBS    Bit 3 = 1       BUFSP
NFSI    Bit 4 = 1       SIS
NFSK    Bit 5 = 1       SKIP
NFBW    Bit 6 = 1       BKWD
NFRS    Bit 7 = 1       REUSE
```

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

From an offset from register 13.

	0	1	2	3	4	
0	Flag 1	A(ECB)				WECB
4	A(Chain Field through EVTABs)					WECH
8	A(Event Variable)					WAEV
C	A(ECBLIST element)					WAEL
10	A(TCA)					WATC
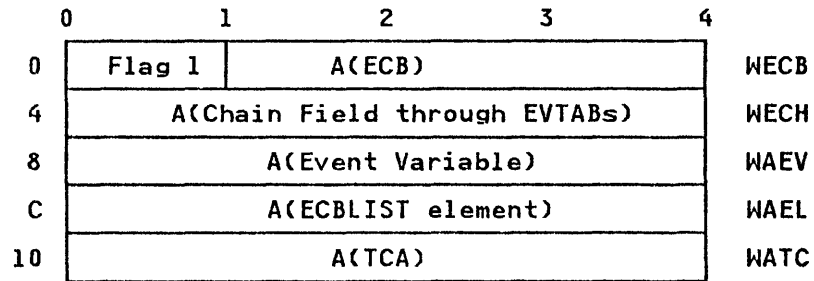
FLAG 1: Bit 0 of the WECB field is set to 1 when an event is complete.

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 the event variable.

How Addressed

As other variables depending on storage class.

	0	1	2	3	4	
0	Flags 1	Flags 2	Status			
4	Anchor for ECB chain					EECH
8	A(DECB) or A(CCB) for I/O					EAEC
C	A(TCA appendage of task for I/O)					ETCA
10	A(DCLCB) or A(FCB) for I/O or A(Called Procedure) for Tasking					EUSI
14	Statement Number					ESND

Flags

FLAGS 1 (EFL0)

ECOM	Bit 0 = 1	Complete
EACT	Bit 1 = 1	Active
EIOF	Bit 2 = 1	I/O EVENT
EDSP	Bit 3 = 1	DISPLAY EVENT
EWIP	Bit 4 = 1	EV has caused entry to an ON-unit
ESNF	Bit 7 = 1	ESNO field contains the statement number

FLAGS 2 (EFL1)

ECHE	Bit 0 = 1	Chain of ECBs exists
EDUM	Bit 1 = 1	Dummy EVENT

EXCLUSIVE BLOCK IOCB (XBI)

Function

    Locks individual records on exclusive files.

When Generated

    By transmitter when required.

How Addressed

    From offset X'24' in IOCB and offset X'14' in the TIA (TEXF).

	0	1	2	3	4	
0	Not Used					XITK
4	XILA	XILB	XILC			
8	XIQE	XIQL	XIQS	XIQC		
C	A(QNAME)					XIQA ->XIQT
10	A(RNAME)					XIRA
14	QNAME = 'SYSIBMIO' (8 bytes)					XIIO
1C	RNAME of Volume Serial Number (6 bytes)					XIVS
24	RNAME of DSNAME (44 bytes)					XIDS
4C			Region Number			XIRN
50	Region No. (cont.)					XIKY
	Space for Record Key (256 bytes)					
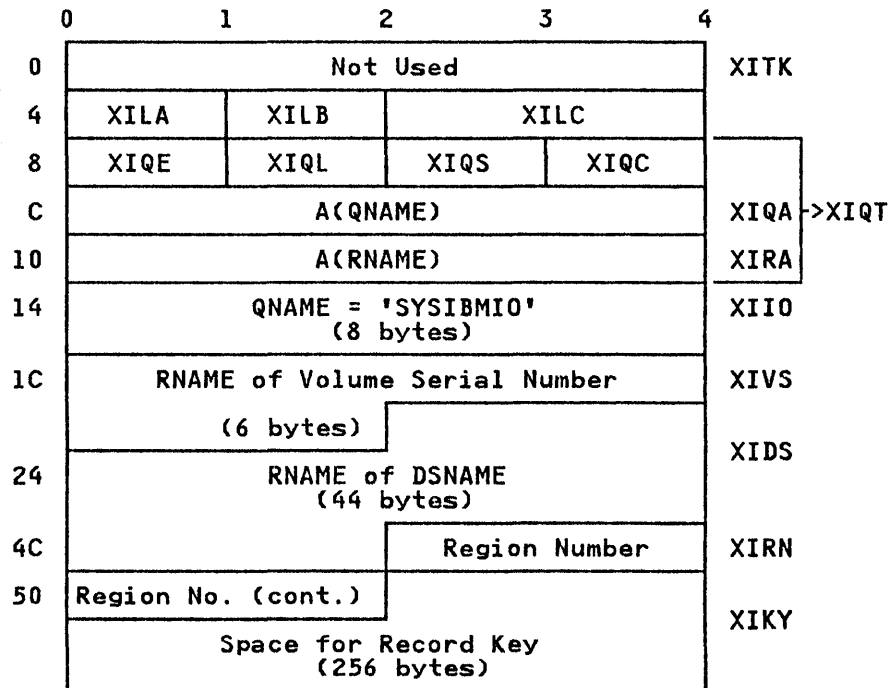
FLAGS

XILA Reserved

XILB Reserved

XILC Reserved

XIQT: This area is a 12 byte field comprised of:

XIQE End marker for enqueue list

XIQL Length of RNAME (XFQL)

XIQS System flags (XFQL) These flags must be set to X'41'.

XIQC Return code from system

XIQA A(QNAME)

XIRA A(RNAME)

XIRN: Region number (in binary right adjusted)

XIKY: RNAME of key

Length of XIKY is keylength of data set restricted such that:

VOL SER||dsname||key ⁄255 ISAM
                     ╲251 regional

EXCLUSIVE BLOCK FILE (XBF)

Function

Identifies data set when locking for exclusive I/O.

When Generated

By the open routine.

How Addressed

From offset X'74' in FCB.

	0	1	2	3	4	
0	A(TCA)					XFTK
4	Flags		XFIL			
8	XFQE	XFQL	XFQS	XFQC		
C	A(QNAME) XFIO					XFQA ->XFQT
10	A(RNAME) XFVS					XFRA
14	QNAME ('SYSIBMIO') (8 bytes)					XFIO
1C	RNAME of Volume Serial Number (6 bytes)					XFVS
20	RNAME of DSNAME (44 bytes)					XFDS
4C	RNAME of Key					XFKY
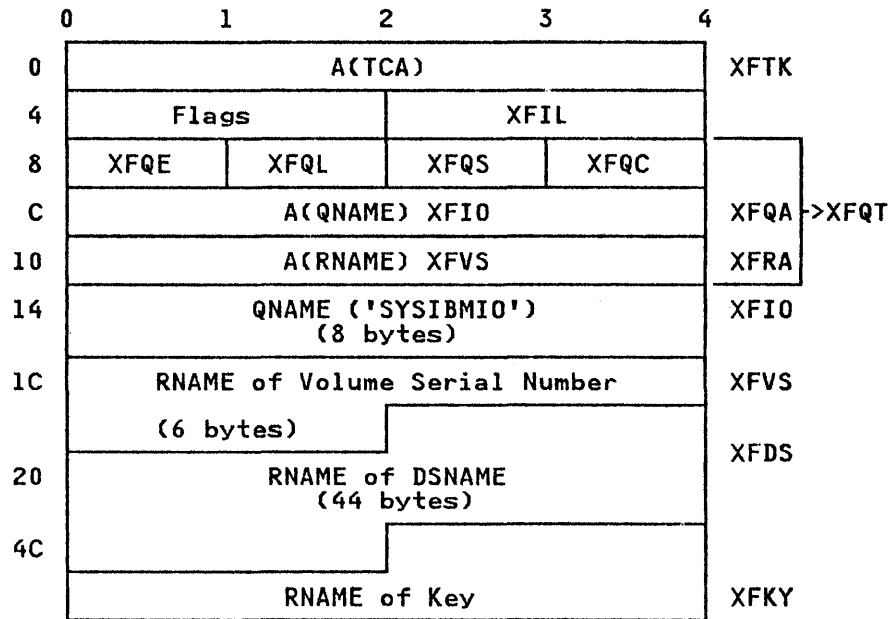
First two words reserved

FLAG (XFLA)

XLOC	Bit 0 = 1	Locked
XNDQ	Bit 1 = 1	No DEQ required
	Bits 2-15	Not used

XFIL: Length of exclusive block attached to IOCB

XFQT: This area is an enqueue list comprised of the following fields:

XFQE End marker for list (X'FF')

XFQL Length of RNAME

XFQS System flags must be X'41'

XFQC Return code from system

XFQA A(QNAME)

XFRA A(RNAME)

XFKY: Length of XFKY is keylength of data set restricted such that:

```
VOL SER ||dsname||key  ⁄255 ISAM
                       ⧹251 regional
```

FILE CONTROL BLOCK (FCB)

Function

Used to access all file information. Contains addresses of the ENVB, DTF, filename, etc.

When Generated

By the open routines during execution.

Where Held

In subpool 1.

How Addressed

From two byte PRV offset which is held at offset X'2' in DCLCB. The PRV address is held at offset X'4' in the TCA.

Common Section

The common section is followed by either the RECORD or STREAM sections.

Offset						Field
	0	1	2	3	4	
-8	Eyecatcher					
0	Statement Mask					FFST
8	A(Invalid Statement Module)					FAIS
C	A(Library Transmitter)					FATM
10	A(DCLCB)					FADL
14	A(DCB) or A(ACB)					FADB or FACB
18	A(Open File Chain)					FAFO
1C	A(data management for in-line I/O)					FAIL
20	Error Bytes		Not Used			
24	FATA	FATB	FATC	Not used		
28	FFLA	FFLB	FFLC	FFLD		
2C	FFLE	FFLF	FFLG	FFLH		
30	Blocksize		Keylength			
34	Record length					FRCL
38	A(First Free IOCB) or A(Hidden Buffer for QISAM LOCATE)					FAFR or FREC
3C	FTYP		FLEN			
40	Reserved for the Checkout Compiler					FGAS
44	FBIF	Not Used				
48	Not Used					

STATEMENT MASK (FFST)

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 DEY
26	DELETE EVENT
27	DELETE KEY EVENT
28	UNLOCK KEY
29	WRITE FROM KEYTO
30	WRITE FROM KEYTO EVENT
31-63	Reserved

ERROR BYTES

FER1

```
FTIP    X'02'    Input transmit (data set)
FTOP    X'03'    Output transmit (data set)
FTOM    X'1A'    OMR read error
FTIX    X'1C'    Input transmit (index set)
FTOX    X'1D'    Output transmit (index set)
FTIS    X'1E'    Input transmit (sequence set)
FTOS    X'1F'    Output transmit (sequence set)
```

FER2

```
FFEF    X'01'    End of file
FRVZ    X'04'    Zero length record variable
FRVS    X'05'    Short record variable
FRVG    X'06'    Long record variable
FKCN    X'07'    Key conversion in character string
FKDP    X'08'    Key duplication
FKSQ    X'09'    Key sequence
FKSP    X'0A'    Key specification (null key)
FKNF    X'0B'    Key not found
FKNS    X'0C'    No space for keyed record
FNIO    X'0D'    No IOCB available
FEAC    X'0E'    Active event
FEUP    X'0F'    No prior read before rewrite
FENC    X'10'    No completed read before rewrite
FETO    X'11'    Permanent output error
FRR2    X'12'    Zero length record read
FEOL    X'13'    Record referenced outside data set
FEXX    X'14'    Unidentified I/O error
FEIR    X'15'    Incomplete read for update
FKTP    X'16'    TP term address specification
FEXS    X'17'    Different FCB same record request
FKCB    X'18'    Key conversion (negative BINARY number)
FKSF    X'19'    Key specification (X'FF' etc)
FASQ    X'1B'    I/O sequence error
FESY    X'20'    Synad error encountered
FRVX    X'21'    Record length < KEYLEN + RKP
FERH    X'22'    Record already held
FEVN    X'23'    Record on non-mounted volume
FESP    X'24'    Data set cannot be extended
FEVS    X'25'    No virtual storage for VSAM
FKNR    X'26'    No keyrange for insertion
FENP    X'27'    No positioning for sequential read
FEUN    X'28'    Attempt to reposition failed
FEST    X'29'    Statement number for data set exceeded
FIEU    X'2A'    Index upgrade error
FEMP    X'2B'    Maximum number of index PTRs
FEIP    X'2C'    Invalid index PTRs
FESW    X'2D'    Invalid sequential write
```

FTYP: 6th and 7th characters of library transmitter name

FLEN: Length of FCB (including DCB)

FATA

```
FDBG    Bit 0 = 1    Open SYSPRINT for error message
FSYS    Bit 1 = 1    SYSPRINT
FCTR    Bit 2 = 1    Reserved for Checkout Compiler
FSTR    Bit 3 = 1    String operation
        Bit 4 = 1    Not used
FDSP    Bit 5 = 1    DISPLAY
FRIO    Bit 6 = 1    RECORD
FSIO    Bit 7 = 1    STREAM
```

FATB

FBAK	Bit 0 = 1	BACKWARDS
FUPD	Bit 1 = 1	UPDATE
FOUT	Bit 2 = 1	OUTPUT
FIPT	Bit 3 = 1	INPUT
	Bit 4 = 1	Not used
FTRA	Bit 5 = 1	TRANSIENT
FDIR	Bit 6 = 1	DIRECT
FSEQ	Bit 7 = 1	SEQUENTIAL

FATC

	Bit 0 = 1	Not used
FEGS	Bit 1 = 1	GRAPHIC option of the ENVIRONMENT attribute
FAXS	Bit 2 = 1	Axes
FPRT	Bit 3 = 1	PRINT
FXCL	Bit 4 = 1	EXCLUSIVE
FKYD	Bit 5 = 1	KEYED
FUNB	Bit 6 = 1	UNBUFFERED
FBUF	Bit 7 = 1	BUFFERED

FFLA

FFIX	Bit 0 = 1	F-format
FVAR	Bit 1 = 1	V-format
FUND	Bit 2 = 1	U-format
FBLO	Bit 3 = 1	Blocked
FSPA	Bit 4 = 1	Spanned
	Bits 5 & 6	Not used
FKLC	Bit 7 = 1	Key in record variable KEYLOC

FFLB

FCON	Bit 0 = 1	CONSECUTIVE
FIND	Bit 1 = 1	INDEXED
FRG1	Bit 2 = 1	REGIONAL(1)
FRG2	Bit 3 = 1	REGIONAL(2)
FRG3	Bit 4 = 1	REGIONAL(3)
FTMP	Bit 5 = 1	TP(M)
FTPR	Bit 6 = 1	TP(R)
FOTH	Bit 7 = 1	Other organization

FFLC

FQSM	X'00'	QSAM
FBSM	X'04'	BSAM
FBSL	X'08'	BSAM (Load)
FQTM	X'0C'	QTAM
FQIS	X'10'	QISAM
FBIS	X'14'	BISAM
FBDM	X'18'	BDAM
FVSM	X'1C'	VSAM

FFLD

FPPT	Bit 0 = 1	Paper tape
FPRI	Bit 1 = 1	Printer
FURD	Bit 2 = 1	Unit record device
FTRM	Bit 3 = 1	The foreground terminal
FEFL	Bit 4 = 1	ENDFILE module loaded
FPHB	Bit 5 = 1	Possible hidden buffer
FEML	Bit 6 = 1	Error module loaded
FGKY	Bit 7 = 1	Genkey

FFLE

FFER	Bit 0 = 1	I/O error
FERI	Bit 1 = 1	Permanent input error
FERO	Bit 2 = 1	Permanent output error
FEOF	Bit 3 = 1	End of file
FHID	Bit 4 = 1	Hidden buffer in use
FEOD	Bit 5 = 1	Move required
FFNV	Bit 6 = 1	Non-SCALARVARYING
FSTK	Bit 7 = 1	Not used

FFLF

FPRD	Bit 0 = 1	Previous READ
FPRS	Bit 1 = 1	Previous READ SET
FPLC	Bit 2 = 1	Previous LOCATE
FPRW	Bit 3 = 1	Previous REWRITE
FPOP	Bit 4 = 1	Previous OPEN or READ IGNORE
FCLS	Bit 5 = 1	Close in progress
FICL	Bit 6 = 1	Implicit close
FRSL	Bit 7 = 1	Previous OPEN (resume load) or READ IGNORE(0)

FFLG

FEPG	Bit 0 = 1	ENDPAGE
FEEX	Bit 1 = 1	End of extent
FCOP	Bit 2 = 1	COPY option active
	Bit 3	Not used
	Bits 4 & 5	Reserved for the Checkout Compiler
FVPF	Bit 6 = 1	Newly opened print file
FNOC	Bit 7 = 1	File not to be closed

FFLH

FILF	Bit 0 = 1	In-line I/O
FILL	Bit 1 = 1	In-line LOCATE
FHYP	Bit 2 = 1	Hyphen at the end of the line
FRGT	Bit 3 = 1	Retry get after concatenation
FCLU	Bit 4 = 1	Current line unfinished
FSPL	Bit 5 = 1	Initial call from IBMBSPL or blanks at the end of record
FBER	Bit 5 = 1	Blanks at the end of record
FNBW	Bit 6 = 1	New buffer wanted
FGPI	Bit 7 = 1	GET prompt issued — input

BUILTIN FUNCTION BYTE (FBIF)

FSKY	Bit 0 = 1	Samekey flag
	Bits 1-7	Not used

Offsets are from start of the FCB.

	0	1	2	3	4	
4C		A(Last IOCB Used) or A(Dummy Buffer for LOCATE)				FALU or FCDA
50		A(first IOCB to be Checked) (BSAM)				FACK
54		Static Chain of IOCBs (BDAM/DISAM/DSAM/VSAM)				FIOC
58		A(IOCB for Last Completed Read)				FALR
5C	FEMT	FEFT	FRET	FAFB		
60		A(error module) When Loaded				FERM
64	FGAM or FFNC	FFLV or FFNF	KEYLOC-1 VSAM or Decrementing Line Count			
68		Record Count				FCCT
6C		A(Dummy Key Area)				FAKY
70		Size of IOCB (BDAM/BISAM) or Current Relative Block (BSAM)				FIOS or FREL
74		A(Exclusive Block FILE)				FXBA
78		Offset Table Used in Record Checking				FRTB
7C		Base OPTCD for RPL (VSAM)				FOPT
80		A(FCB) or A(FAFB)				FAWB

FEMT: 7th character of the error module name

FEFT: 7th character of the endfile module name

FRET: Data management return code (regional output)

FAFB: Work byte for associated files

FFNC: Function byte

FARF	Bit 0 = 1	READ file
FAPF	Bit 1 = 1	PUNCH file
FAWF	Bit 2 = 1	PRINT file
FOMR	Bit 3 = 1	OMR (no other lists on)
FRFI	Bit 4 = 1	R in FUNC option
FPFI	Bit 5 = 1	P in FUNC option
FPWI	Bit 6 = 1	W in FUNC option
FASC	Bit 7 = 1	Associated file

FFLV: VSAM flags

FKSD	Bit 0 = 1	KSDS
FESD	Bit 1 = 1	ESDS
FRDS	Bit 2 = 1	RRDS
FPTH	Bit 3 = 1	ALTERNATE INDEX PATH
FNUM	Bit 4 = 1	ALTERNATE INDEX PATH (non-unique)

```
FSKP    Bit 5 = 1      SKIP
        Bit 6 = 1      Not used
FPLO    Bit 7 = 1      Position lost


FCNF:   Conflict byte

FPII    Bit 0 = 1      Prior READ invalid
FPPI    Bit 1 = 1      Prior PUNCH invalid
FPWI    Bit 2 = 1      Prior PRINT invalid
FPLI    Bit 3 = 1      Prior PRINT last line invalid
        Bit 4-7        Not used
```

Stream I/O Section

Offsets are from the start of the FCB.

```
        0           1           2           3           4
      ┌───────────────────────────────────────────────┐
  4C  │      A(Next Available Byte in a Buffer)        │   FCBA
      ├───────────────────────┬───────────────────────┤
  50  │ Bytes Remaining in    │   Value of Count       │
      │      Buffer           │  Built-in Function     │
      ├───────────────────────┼───────────────────────┤
  54  │      Page Size        │      Line Size         │
      ├───────────────────────┼───────────────────────┤
  58  │   Current Line No.    │     Buffer Size        │   FMAX
      ├───────────────────────┴───────────────────────┤
  5C  │       A(Copy Position in Buffer)               │   FCPM or FNTP
      │                   or                           │
      │     A(Next TPUT Position) for OUTPUT           │
      ├───────────────────────────────────────────────┤
  60  │       A(DCLCB for COPY file)                   │   FCPF
      ├───────────────────────────────────────────────┤
  64  │       A(Copy Module Input                      │   FCPA or FTAB
      │                   or                           │
      │       A(Tab Module Output)                     │
      ├───────────────────────────────────────────────┤
  68  │          Record Count                          │   FRCT
      ├───────────────────────────────────────────────┤
  6C  │            F(SIOCB)                            │   FSCB
      └───────────────────────────────────────────────┘
```

FETCH CONTROL BLOCK (FECB)

Function

The FECB is used to contain information about modules specified in FETCH statements.

How Addressed

FECBs are chained together. The chain starts in field TFEP, which is held in the TIA at offset X'3C'

Where Held

FECBs are set up by IBMBPFR in non-LIFO storage.

When Generated

When a module is fetched.

0	1	2	3	4	
0	Chain Field				ZFCH
4	PRV Offset				ZFPO
8	Name of Module (8 bytes)				ZFNM
10	AMODE Switching Code				ZTRFCDE
20	A(Fetched module entry point)				ZTARGET
24	A(Call R14 Save Area)				ZSAVR14

FLOW STATEMENT TABLE

Function

Used to implement the compiler FLOW option. Holds the last 'n'
statement number pairs and the last 'm' procedure names
executed. ('n' and 'm' are programmer defined.)

When Generated

Storage is allocated during initialization if the FLOW option
has been specified. The table is continually updated as the
program is executed.

Where Held

In initial storage area.

How Addressed

From offset X'4C' in the TCA.

	0	1	2	3	4	
0	Code to access IBMBEFLA to initialize the flow table for subtasks. Called when bit 6 in AFLF is set.					ARGT
10	Total length of the table					AFLL
14	A(next free field in statement number section)					ANEN
18	A(start of names section of table)					AASB
1C	A(next free field in names section)					ANEB
20	A(end of table)					AAEB
24	A(start of number section)					ASBS
28	Flag Byte					AFL1

FLAG BYTE (AFL1)

ANON	Bit 0 = 1	No statement numbers requested in flow trace, for example FLOW(0,20)
AFLI	Bit 1 = 1	Last entry was IN
AILF	Bit 2 = 1	Flow to be done inline if last entry was in; flow cannot be done if last entry was out.
AINT	Bit 3 = 1	Interrupt not recorded
AGOT	Bit 4 = 1	GOTO-out-of-block has occurred
	Bits 5-7	Not used

```
       0         1         2         3         4
      ┌─────────┬─────────┬───────────────────┐
      │AFLF Flag│AFLG Flag│     Statement     │
      ├─────────┴─────────┼─────────┬─────────┤
      │      Number       │AFLF Flag│AFLG Flag│
      ├───────────────────┴─────────┴─────────┤
      │          Statement Number              │
      ├────────────────────────────────────────┤
      │                                         │
      ├────────────────────────────────────────┤       ASBD
      │     Names of blocks truncated to        │
      │            8 characters                 │
      ├────────────────────────────────────────┤
      │                                         │
      └────────────────────────────────────────┘
```

INFORMATION BYTE (AFLF)

ATBI	Bit 0 = 1	Branch-in entry
ABCD	Bit 1 = 1	BCD form for this entry
AXTX	Bit 2 = 1	BCD in text reference form
ADUM	Bit 3 = 1	Dummy entry after ON-unit exit
ACHK	Bit 4 = 1	Statement number in TSO line number
AGTO	Bit 5 = 1	Entry before GOTO call (old flag)
ATKC	Bits 6 = 1	The next in entry is in a new task
	Bit 7	Not used

TASKING INFORMATION BYTE (AFLG)

ADES	Bit 0 = 1	Task being descheduled
ARES	Bit 1 = 1	Task being rescheduled
ATRM	Bit 2 = 1	Task being terminated
	Bits 3-7	Not used

INTERLANGUAGE ROOT CONTROL BLOCK (IBMBILC1)

Function

Connects ZCTL and interlanguage VDA to interlanguage 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

Address generated by linkage editor.

```
       0         1         2         3         4
   ┌─────────────────────────────────────────┐
 0 │              "Eye Catcher"               │  ZILCE
   ├─────────────────────────────────────────┤
 4 │            Address of ZCTL               │  ZICT
   ├────────┬────────┬──────────┬─────────────┤
 8 │  ZICF  │  ZIFF  │ Not Used │    ZITF      │
   └────────┴────────┴──────────┴─────────────┘
```

FLAGS

ZICF Bit 0 = 1   Indicates COBOL is active in program
ZIFF Bit 0 = 1   Indicates a procedure which called FORTRAN is active
ZITF Bit 0 = 1   Indicates a task is accessing IBMBILC1

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

From offset X'0' in ZCTL.

	0	1	2	3	4	
0	"Eye Catcher"					ZVDAE
4	A(Previous Interlanguage VDA or Zero)					ZPVD
8	Flag1	Flag2	Flag3	Pgm. Mask		
C	A(Current DSA)					ZPDR
10	A(Caller's PICA or ESPIE Parm List					ZPCPL
14	Saved Language's Macro Type					ZPCPM
18	Saved Language's Token					ZPCPT
1C	18 Word Imitiation Save Area					ZPSP

FLAG1 (ZPRP)

Bit 0 = 1   If there is a previous call to COBOL
Bit 1 = 1   If there is a previous call to FORTRAN
Bit 2 = 1   If main procedure is not PL/I

FLAG2 (ZPFN): This is a nonrecurring flag.

FLAG3 (IEUAX): This flag when set on, indicates the type of language.

INTERRUPT CONTROL BLOCK (ICB)

Function

    Acts as a parameter list to IBMBERR.

When Generated

    After an error has been detected.

Where Held

    As a VDA in the LIFO stack.

How Addressed

    Passed as parameter list to IBMBERR addressed by register 1.

```
          0          1          2          3          4
     ┌────────────────────────────────────────────────┐
   0 │                  Error code                      │  HLCD
     ├────────────────────────────────────────────────┤
   4 │               Condition Qualifier                │  HLQU
     ├───────────┬────────────────────────────────────┤
   8 │ DSA Level │             Flags                    │
     ├───────────┴────────────────────────────────────┤
   C │              A(Array Element)                    │  HLEA
     ├────────────────────────────────────────────────┤
  10 │                 A(SYMTAB)                        │  HLSY
     ├────────────────────────────────────────────────┤
  14 │ A(Pointer for BASED or CONTROLLED Var.)          │  HLPT
     └────────────────────────────────────────────────┘
```

Note: For unqualified errors, only field HLCD is passed.

HLQU

Condition qualifier	= A(DCLCB) for I/O condition
	= A(CSECT) for CONDITION condition
	= A(SYMTAB) for CHECK condition
	= A(SYMTAB LIST)

FLAGS (HLFG)

HLFZ	Bit 0 = 1	Use the address in HLSY for data directed I/O
HLFY	Bit 1 = 1	Element address in list
HLFX	Bit 2 = 1	CHECK enablement unknown
HLFW	Bit 3 = 1	Qualifier is address of SYMTAB list
HLFV	Bit 4 = 1	Use word 6 to address the generation of variable being checked
HLFU	Bit 5 = 1	Do not print or display data
	Bits 6-7	Not used

INPUT/OUTPUT CONTROL BLOCK (IOCB)

Function

Used as a data management parameter list during certain record I/O statements and to hold information about statement type during the time between a record I/O statement and the associated WAIT statement.

When Generated

Either by the PL/I transmitter module (BISAM or BDAM) or by OPEN.

Where Held

In-non-LIFO storage for VSAM, in subpool 0 for BSAM (obtained by GETPOOL), BISAM or BDAM (obtained in subpool 1 for non-multitasking, in subpool 0 for tasking).

How Addressed

By fields in the FCB. IOCBs are chained together and the actual field used to address them depends on the type of statement being executed.

| Common Section

```
        0         1         2         3         4
   ┌─────────────────────────────────────────────┐
 0 │           Static Forward Chain              │  ICHN
   ├─────────────────────────────────────────────┤
 4 │      Chain of Free or Unchecked IOCBs       │  INXT or IRGN
   │                    or                       │
   │     Region Number, Left Adjusted (BDAM)     │
   ├──────────┬──────────┬───────────────────────┤
 8 │   IFLA   │   IFLB   │   Error Codes (IERR)   │
   ├──────────┴──────────┴───────────────────────┤
 C │           Request Control Block             │  IRCB
   ├─────────────────────────────────────────────┤
10 │   1st Word of Record Descriptor; A(RCD)     │  IORD
   ├─────────────────────────────────────────────┤
14 │       2nd Word of Record Descriptor;        │  IORL
   │          Flags and Record Length            │
   ├─────────────────────────────────────────────┤
18 │          1st Word of Key Descriptor         │  IOKD or IFNA  ┐
   ├─────────────────────────────────────────────┤                ├─>IREF
1C │          2nd Word of Key Descriptor         │  IOKL or IFBK  ┘
   ├─────────────────────────────────────────────┤
20 │             A(EVENT Variable)               │  IEVT
   └─────────────────────────────────────────────┘
```

FLAG BYTE (IFLA)

IFXV	Bit 0 = 1	Record locked
IFMU	Bit 1 = 1	Record to move flag
IFSU	Bit 2 = 1	Varying string with non-scalar variable
IFUS	Bit 3 = 1	IOCB in use
IFER	Bit 4 = 1	General error flag
IFDR	Bit 5 = 1	Dummy records are being printed or displayed
IFDB	Bit 6 = 1	Dummy buffer acquired
IFCH	Bit 7 = 1	IOCB checked

FLAG BYTE (IFLB): Code byte containing offset within 'look-up' table used for record checking

ERROR CODES (IERR)

IEOF	X'01'	End of file
ITID	X'02'	Input transmit
ITOP	X'03'	Output transmit
IRVZ	X'04'	Zero length record variable
IRVS	X'05'	Short record variable
IRVG	X'06'	Long record variable
IKCN	X'07'	Key conversion
IKDP	X'08'	Key duplication
IKSQ	X'09'	Key sequence
IKSP	X'0A'	Key specification
IKNF	X'0B'	Key not found
IKNS	X'0C'	No space for keyed record
INIO	X'0D'	No IOCB available
IEAC	X'0E'	Active event
IEUP	X'0F'	No prior READ before REWRITE
IENC	X'10'	No completed READ before REWRITE
IETO	X'11'	Permanent output error
IRRZ	X'12'	Zero length record read
IEOL	X'13'	Record reference outside data set
IEXX	X'14'	Unidentified IO error

IOKL: Flags and key length

IREF: Relative block or record number (2 words) (BDAM)

IFNA: Next address feedback (BDAM spanned)

IFBK: BDAM feedback (BDAM spanned)

Non-VSAM Section

This section starts at offset X'24'.

```
        0         1         2         3         4
     ┌─────────────────────────────────────────────┐
24   │      A(ECB) for Regional Sequential Only     │   IADE or IXLV or IRLB
     │                     or                       │
     │      A(Exclusive Block) for Direct Only      │
     │                     or                       │
     │   A(Binary Region No.- Regional(1) Update    │
     ├─────────────────────────────────────────────┤
28   │          A(Implementation Appendage)         │   ITIA
     └─────────────────────────────────────────────┘
```

DATA MANAGEMENT EVENT CONTROL BLOCK

```
        0         1         2         3         4
     ┌─────────────────────────────────────────────┐
2C   │ BDAM Exception Codes in 2nd & 3rd Bytes      │   IECB
     ├───────────────────────┬─────────────────────┤
30   │  I/O Operation Type   │    Record Length    │
     │  Set by Data Mgmt.    │       (ILEN)        │
     ├───────────────────────┴─────────────────────┤
34   │          A(Data Control Block)               │   IDCB
     ├─────────────────────────────────────────────┤
38   │       A(Buffer) or A(Record Variable)        │   IREC
     ├─────────────────────────────────────────────┤
3C   │   A(Status Indicators) (BSAM & BDAM) or      │   ISTS or
     │          A(Logical Record)                   │   ILOG
     ├─────────────────────────────────────────────┤
40   │          A(Dummy Buffer) (BSAM)              │   IADB
     │                     or                       │   or
     │  A(Next Record Feedback)───>IREF (BSAM)      │   INLF
     │                     or                       │   or
     │          A(KEY) (BDAM and BISAM)             │   IKEY
     ├─────────────────────────────────────────────┤
44   │   A(Relative Block or Record) that is,       │   IBLK
     │          A(IREF) (BDAM) or                   │   or
     │          BISAM Exception Codes               │   IEXI
     ├─────────────────────────────────────────────┤
48   │  A(Next Record Feedback)───>IREF (BDAM)      │   INDF
     │                     or                       │   or
     │   Start of Any Appended Buffer (BSAM)        │   ISBF
     ├─────────────────────────────────────────────┤
4C   │      Start of Any Appended Buffer            │   IDBF
     │          (BDAM-or-BISAM)                     │
     └─────────────────────────────────────────────┘
```

VSAM Section

This section also starts at offset X'24'.

```
        0         1         2         3         4
     ┌─────────────────────────────────────────────┐
24   │            A(Dummy Buffer)                   │   IDUB
     ├─────────────────────────────────────────────┤
28   │            A(First Key Area)                 │   IKSV
     ├─────────────────────────────────────────────┤
2C   │            A(Second Key Area)                │   IKST
     ├─────────────────────────────────────────────┤
30   │        Pointer for LOCATE Requests           │   IPTR
     ├─────────────────────────────────────────────┤
34   │                A(ONKEY)                      │   IONK
     └─────────────────────────────────────────────┘
```

DATA MANAGEMENT EVENT CONTROL BLOCK

Offset	0 1 2 3 4	Name
38	A(Data Management Event Control Blocks)	IEVC
3C	A(Request Parameter List) SHOWCB Parameter List	IRPL
40	A(Header)	ISHD
44	A(Element)	ISEL
48	Type Codes	ISTC
4C	A(Block)	ISBL
50	A(User Area)	ISAR
54	Length of User Area	ISLN
58	Element Codes	ISEC
5C	User Area MODCB Parameter List	ISUA
60	A(Header)	IMHD
64	A(Element) Maximum of 3	IMEL
68	A(Element)	
6C	A(Element)	
70	MODDCB Type Codes	IMTC
74	A(Block)	IMBL
78	Not Used	IM2C
7C	Area	IARA
80	Not Used	IM2D
84	Area Length	IARL
88	Not Used	IM30
8C	Key Length	IKYL
90	Not Used	IM34
94	OPT Code	IOPT
98	Not Used	IM35
9C	Record Length	IRCL

Element control entries start at offset X'78' and continue to end of IOCB. Each entry occupies 2 words, with keyword type code set in 1st half-word as follows:

    IMab=X'00ab'

For VSAM files, the IOCB has an associated appendage, comprising the RPL, a dummy buffer if the file has the BUFFERED attribute, and a key save area if the data set is a VSAM KSDS.

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.

How Addressed

From an offset from register 3 known to compiler code for
internal keys.

0	1	2	3	4	
0	A(Key String)				VKDA
4	Flag	Not Used	Length of Key		VKDL
8	Region No. in Fixed Binary, Right Adj.				VKDR
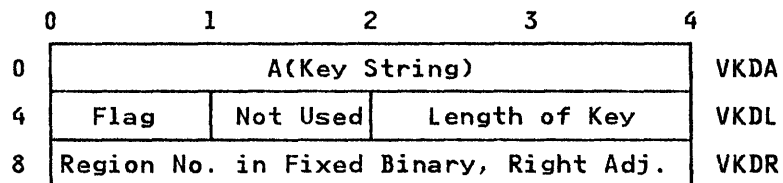
VKDA: The address of the source key (excluding the length bytes
if VARYING)

FLAG (VKDV)

VKFV	Bit 0 = 1	KEYTO string is VARYING. (If this bit is set, the I/O transmitters will set the current length field).
VKFB	Bit 1 = 1	This bit is set when the VKDR field contains a region number.
	Bits 2-7	Not Used

VKDL: Length of key string (excluding length bytes for
VARYING); current length for KEY or KEYFROM, maximum length for
KEYTO.

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 depending on storage class
Label temporaries	When required for GOTO to label constant

Where Held

Depends on the storage class of the data item.

How Addressed

As a variable.

Label Variable and Label Temporary

```
     0         1         2         3         4
   +-+-----------------------------------------+
 0 | | A(Label Constant) Assigned to the       |
   | |         Label Variable                  |
   +-+-----------------------------------------+
 4 | | A(DSA) at the Time of Assignment of     |
   | |          Owning Block                   |
   +-+-----------------------------------------+
```

Word 1: bit 0 = 0 Address of label
                = 1 Text reference

Word 2: bit 0 always = 0

Label Constant

```
     0         1         2         3         4
   +-------------------------------------------+
 0 |                A(Label)                   |
   +-------------------------------------------+
 4 | Value to be loaded into Reg. 2 on GOTO    |
   | It becomes the new base register.         |
   +-------------------------------------------+
```

LIBRARY WORKSPACE (LWS)

Function

Space reserved for two preformatted 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

From offset X'48' in each DSA.

```
        0          1          2          3          4
    ┌───────────────────────┬───────────────────────┐  ┐
  0 │      DSA Flags        │  Offset to the ONCA    │  │
    ├───────────────────────┴───────────────────────┤  │
  4 │          The same back-chain and              │  │
    │   register save areas that are shown           │  ├─>LLW0
    │              in the DSA                        │  │
    │        (See the DSA Control Block)            │  │
    │                                                │  │
    ├────────────────────────────────────────────────┤  │
 50 │             56 Byte Workspace                 │  ┘
    ├───────────────────────┬───────────────────────┤  ┐
 88 │      DSA Flags        │  Offset to the ONCA    │  │
    ├───────────────────────┴───────────────────────┤  │
 8C │          The same back-chain and              │  │
    │   register save areas that are shown           │  ├─>LLW1
    │              in the DSA                        │  │
    │        (See the DSA Control Block)            │  │
    │                                                │  │
    ├────────────────────────────────────────────────┤  │
 D8 │             56 Byte Workspace                 │  ┘
    ├────────────────────────────────────────────────┤
110 │              Current ONCA                     │
    │        (see the ONCA Control Block)           │
    └────────────────────────────────────────────────┘
```
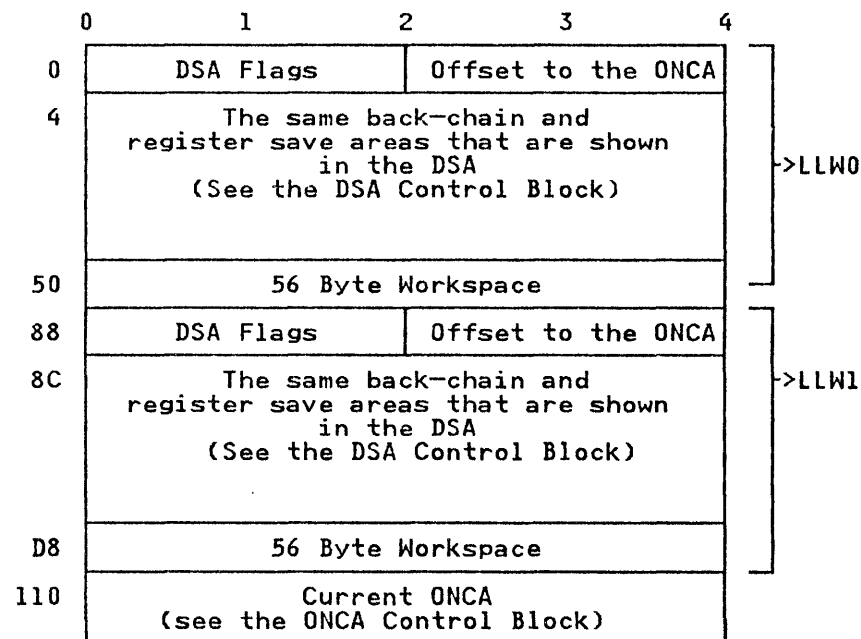
DSA FLAGS: These flags are the same as Flag Byte 0 and Flag Byte 1 in the DSA. For further information on these flag bytes and their contents, see "Flags" on page 348.

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 from the current generation of library workspace. The offset is held as a halfword at offset X'2' in LWS.

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 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, "Error and Condition Handling" on page 105.)

```
            0         1         2         3         4
        ┌─────────────────────────────────────────┐
   0    │      Chain Back to Previous ONCA         │   LOCB
        ├──────────────────────┬──────────┬────────┤
   4    │       ONCODE         │Flag LFG1 │Not Used│
        ├──────────────────────┴──────────┴────────┤
   8    │      String Locator for ONFILE           │   LOFL
        │              (8 bytes)                    │
        ├───────────────────────────────────────────┤
  10    │      String Locator for ONCHAR           │   LOCH
        │              (8 bytes)                    │
        ├───────────────────────────────────────────┤
  18    │      String Locator for ONSOURCE         │   LOSC
        │              (8 bytes)                    │
        ├───────────────────────────────────────────┤
  20    │      String Locator for ONKEY            │   LOKY
        │              (8 bytes)                    │
        ├───────────────────────────────────────────┤
  28    │      String Locator for DATAFIELD        │   LODF
        │              (8 bytes)                    │
        ├───────────────────────────────────────────┤
  30    │      String Locator for ONIDENT          │   LOID
        │              (8 bytes)                    │
        ├───────────────────────────────────────────┤
  38    │      A(Record I/O EVENT Variable)        │   LEVT
        ├───────────────────────────────────────────┤
  3C    │          Pointer for ONATTN              │   LPAT
        ├───────────────────────────────────────────┤
  40    │             ONCOUNT                      │   LCNT
        ├───────────────────────────────────────────┤
  44    │          Retry Environment               │   LREN
        ├───────────────────────────────────────────┤
  48    │      Retry Address for Conversion        │   LRAD
        ├─────────────┬────────────────┬────────────┤
  4C    │   X'40'     │  X'00000000'   │ Flag LFG3  │
        ├─────────────┼────────────────┼────────────┤
  50    │   LCT1      │  Retry Codes   │ Not Used   │
        └─────────────┴────────────────┴────────────┘
```

FLAG (LFG1)

LFOF	Bit 0 = 1	ONFILE valid
LFOC	Bit 1 = 1	ONCHAR/ONSOURCE valid
LFID	Bit 2 = 1	ONIDENT valid
LFKY	Bit 3 = 1	ONKEY valid
LFDF	Bit 4 = 1	DATAFIELD valid
LFEV	Bit 5 = 1	Asssociate EVENT variable
LFAT	Bit 6 = 1	ONATTN valid
LFCT	Bit 7 = 1.	1 ONCOUNT valid

FLAG (LFG3)

LFSC	Bit 0 = 1	ONSOURCE or ONCHAR is used in an ON-unit
LFSS	Bit 1 = 1	ONSOURCE set in ONCA
	Bits 2-7	Not used

LCT1: Copy of TCA flag byte 1 (TFB1). For further details, see "Task Communication Area (TCA)" on page 407.

RETRY ADDRES (LRAD): The offset from the base of the library module involved to the address where a conversion is attempted again if ONSOURCE or ONCHAR is used.

ON CONTROL BLOCK (ONCB)

Function

Contains pointer to associated ON-unit, or indicates action to be taken when interrupt occurs.

How Addressed

From offset X'60' in the DSA.

When Generated

Static ONCBs are generated during compilation, one for each ON statement. Dynamic ONCBs are generated by the prolog 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 Held

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.

Static and Dynamic ONCBs

Static ONCBs are generated for unqualified conditions. Dynamic ONCBs are generated for qualified conditions (ENDPAGE, ENDFILE, etc.)

Dynamic ONCB

0		1	2	3	4	
0	A(previous dynamic ONCB in block) or zero, if first					LDBC
4	Qualifier					LDQU
8	Condition Code	Flag (LDFG)	Not Used			
C	Target					LDTR

Static ONCB

0		1	2	3	4	
0	Condition Code	Flag (LSFG)	Not Used			
4	Target					LSTR
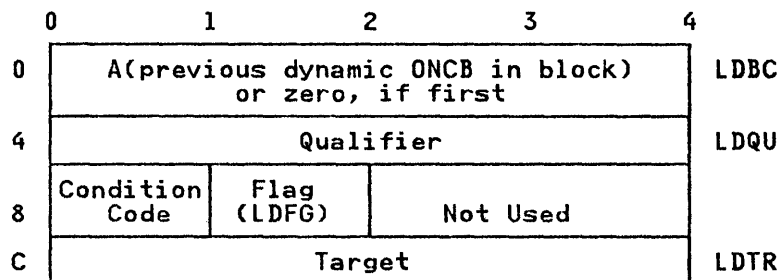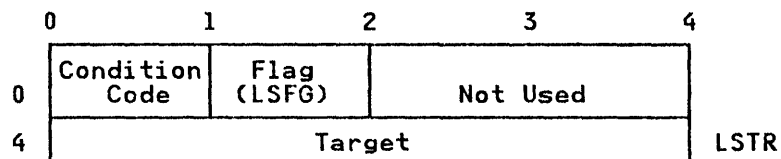
QUALIFIER: A(DCLCB) for I/O conditions A(SYMTAB) for CHECK
A(CSECT) for CONDITION condition

FLAG (LDFG AND LSFG)

```
LSF0   Bit 0 = 1      SYSTEM specified
LSF1   Bit 1 = 1      Null ON-unit
LSF2   Bit 2 = 1      GOTO only ON-unit
LSF3   Bit 3 = 1      Condition established
LSF4   Bit 4 = 1      Not Used
LSF5   Bit 5 = 1      Enabled at block entry
LSF6   Bit 6 = 1      Condition enabled
LSF7   Bit 7 = 1      SNAP specified
```

TARGET: Address of ON-unit, or offset in DSA of word containing
A(label variable)

OPEN CONTROL BLOCK (OCB)

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

From an offset from register 3 known to compiled code.

	0	1	2	3	4	
0	NPAA	NPAB	NPAC	NPAD		
4	Open Conflict Mask					NDEM
8	A(ENVB)					NENV
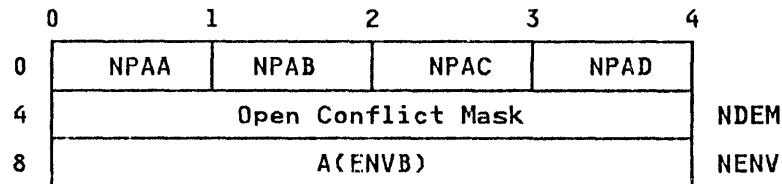
OPEN ATTRIBUTES: This word indicates the explicit and implied attributes on the OPEN statement.

NPPA	80	Debug open of SYSPRINT
	10	reserved (STRING)
	04	DISPLAY
	02	RECORD
	01	STREAM

NPAB	80	BACKWARDS
	40	UPDATE
	20	OUTPUT
	10	INPUT
	08	TRANSIENT
	02	DIRECT
	01	SEQUENTIAL

NPAC	20	AXES
	10	PRINT
	08	EXCLUSIVE
	04	KEYED
	02	UNBUFFERED
	01	BUFFERED

OPEN CONFLICT MASK (NDEM): This is a mask generated by the compiler containing bits for all attributes which conflict with those on the OPEN statement.

ORDERED DELETE LIST (ODL)

Function

Hold list of transient modules to be deleted during program termination.

When Generated

During program initialization.

This block is initialized to binary zeros; each routine places its address in the appropriate field as soon as it is loaded.

Where Held

Program Management area.

How Addressed

From offset X'38' in the TCA.

	0	1	2	3	4
0	A(IBMBEDWA)				
4	A(IBMBEDTA)				
8	A(IBMBKOTA)				
C	A(Extended float simulator)				
10	A(IBMBMYEA)				
14	A(IBMBMCTA)				
18	A(IBMBSPCA)				
1C	A(IBMBPESA)				
20	A(IBMBCCLA)				
24	A(IBMBSTAB)				
28	A(IBMBEIIA)				

PLIMAIN

Function

Holds address of entry point of main procedure.

When Generated

During compilation of procedures with the MAIN option.

Where Held

A separate control section in the load module.

How Addressed

Address resolved by linkage editor.

```
        0           1           2           3           4
      ┌─────────────────────────────────────────────────────┐
    0 │  VCON(Primary Entry Point to Program)                │
      ├─────────────────────────────────────────────────────┤
    4 │                    Zero                              │
      └─────────────────────────────────────────────────────┘
```

DUMMY PLIMAIN: A control section in IBMBPIRA and IBMTPIRA holding addresses of error message module. This control section is link-edited if no compiler generated PLIMAIN exists.

PLISTART PARAMETER LIST

Function

Used to pass housekeeping information extracted by compiler to PL/I initialization routines.

When Generated

PLISTART is a CSECT generated by the compiler for every external compilation. The parameter list is part of the PLISTART CSECT.

General Format of PLISTART

PLISTART contains the three standard entry points PLISTART, PLICALLA, and PLICALLB. When entry is made, addressability is established register 0 pointed at the parameter list and a branch made to entry point A,B, or C of the initialization routine from PLISTART, PLICALLA, and PLICALLB respectively.

The format of the parameter list for PLISTART is given below.

Addressed by register 0

COMMON SECTION

	0 1 2 3 4	
0	A(PLIMAIN)	ZYMA
4	A(SYSPRINT DCLCB)	ZYSP
8	A(PLIFLOW)	ZYFL
C	A(IBMBSTABA)	ZYTB
10	LENGTH of PRV	ZYPR
14	Always set to zero (after Release 2.0)	ZYFG
18	A(Remote Shared Library Module List)	ZYAL
1C	A(PLICOUNT) or zero if NOCOUNT option	ZYCT
20	A(PLIXOPT) or zero if none	ZYXO

NON-EXTENDED ARCHITECTURE SECTION

	0 1 2 3 4	
24	ZYLTR2	A(IBMBPOPTA) if any	ZYPO
28	A(PLIXHD) if any, or zero	ZYHD
2C	A(IBMBEATA) if INTERRUPT option used	ZYEA
30	ZYLTR3	

	0	1	2	3	4	
24	ISA Length for Non-Multitasking					ZONI
28	Non-Multitasking Options Word					ZONW
2C	Major Task ISA Length					ZOTI
30	Minor Task ISA Length					ZOSI
34	Version Number	Not Used	Maximum Number of Subtasks			
38	Tasking Options Word					ZOTW
3C	Flow (ZOFW1)		Flow (ZOFW2)			
40	Heap Initial Size					ZONHI
44	Heap Increment Size					ZONHC
48	ISA Increment Size					ZONIC
4C	Major Task Heap Initial Size					ZOTHI
50	Major Task Heap Increment Size					ZOTHC
54	Major Task ISA Increment Size					ZOTIC
58	Minor Task Heap Initial Size					ZOSHI
5C	Minor Task Heap Increment Size					ZOSHC
60	Minor Task ISA Increment Size					ZOSIC

END MARKERS (ZYLTR2 AND ZYLTR3): Bit 0 of these end markers is set to 1 to designate the end of the compiler for Release 2 and Release 3.

RECORD DESCRIPTOR (RD)

Function

To hold data about the record variable.

When Generated

During Compilation.

Where Held

Static control section.

How Addressed

From an offset from register 3 known to compiled code.

	0	1	2	3	4	
0		A(Record Variable)				VRDA
4	Flag	Length of Data to Transmit				VRDL
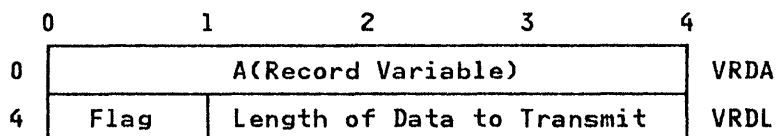
FLAG (VRDV): These bits indicate the type of INTO or FROM argument as follows:

```
VRFF   X'00'  For fixed length strings
VRFA   X'01'  For area variables
VRFV   X'02'  For varying length character strings
VRFB   X'03'  For varying length bit strings
```

LENGTH (VRDL): This field is the 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 current length for output operations and the maximum length for input operations.

REQUEST CONTROL BLOCK (RCB)

Function

Used by the record I/O interface module (IBMBRIOA) to check the validity of an I/O statement. The instruction in RTMI is carried out by IBMBRIOA.

When Generated

During compilation.

Where Held

Static internal control section.

How Addressed

From an offset from register 3 known to compiled code.

0	1	2	3	4

0	REQ1	REQ2	REQ3	Not Used
4	RTMI			
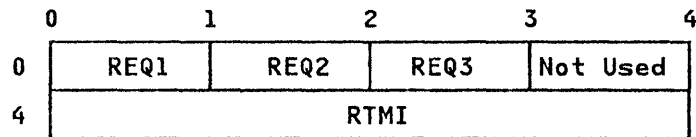
STATEMENT IDENTIFICATION (REQ1)

```
RRED   X'00'   READ
RREW   X'04'   REWRITE
RWRT   X'08'   WRITE
RLOC   X'0C'   LOCATE
RDEL   X'10'   DELETE
RUNL   X'14'   UNLOCK
RWAT   X'18'   WAIT
```

OPTION CODES (REQ2)

```
RNON   X'00'   None
REVN   X'01'   EVENT
RNOL   X'02'   NOLOCK
RIGN   X'20'   IGNORE
RSET   X'40'   SET
RIFR   X'80'   INTRO/FROM
```

KEY OPTION CODES (REQ3)

```
RKFR   X'20'   KEYFROM
RKTO   X'40'   KEYTO
RKFR   X'80'   KEY
```

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.

TM instruction used by IBMBRIOA for testing the validity of a statement.

   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.

When the TM instruction is issued, register 2 points to the File Control Block (FCB) and SSS becomes the appropriate offset in the statement mask, field FFST.

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 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'48' 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'4C' in the TIA.

	0	1	2	3	4	
0	A(Next Table)					ACTB
4	A(Static CSECT of PROCEDURE)					ACST
8	Name of Procedure					ACEP
10	Flags					ACFL
14	A(First Segment)					ACBS
18	A(Next Segment)					ACSG
1C	Number of Entries					ACNG
20	Length of Segment					ACLG
	Count Entry or Number					
	Count Entry					
	Count Entry or Number etc.					

ACBS: The address held in ACBS is the address of ACGS. If tables are segmented, second and subsequent sections of the table will start at a point equivalent to ACSG.

FLAG (ACFL)

ACBI	Bit 0 = 1	The last update was for a branch in
ACGT	Bit 1 = 1	The last update was for a GOTO out of a block
ACIA	Bit 2 = 1	The table is inactive
ACNM	Bit 3 = 1	The table is for a procedure with the GONUMBER option
ACUI	Bit 4 = 1	The table is uninitialized
ACZL	Bit 5 = 1	The table contains unexecuted ranges
	Bits 6 & 7	Not used

STATEMENT NUMBER TABLE

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 or GONUMBER option is in effect.

Where Held

Static internal control section.

How Addressed

From offset X'8' from entry point of main procedure.

Sections of Table

Because offsets are held in two bytes and the value may in fact take up to three bytes, it is necessary to hold the table in sections.

Statement Number Format

Halfword binary right-aligned.

0	1	2	3	4	
0	A Primary Entry Point of Block				ZMEP
4	Size of Code Generated for Block (in bytes)				ZSBL
8	A(End of First Section or Start of Second Section)				ZANB
C	Offset		Statement Number		
	Offset		Statement Number		
	A(End of Second Section/Start of Third Section)				
	Offset		Statement Number		
	Etc.				

Line Number Format

When line numbers are generated they are held in 6-byte fields. The first 27 bits hold the line number, right adjusted in binary. The last five bits hold the number of the statement on the line, again right adjusted in binary.

The presence of line numbers is indicated by bit 5 of Flags 2 in the DSA being set to 1. The validity of Flags 2 is indicated by bit 15 in the flags in the first two bytes of the DSA being set

to 1. The presence of line numbers is indicated if <u>both</u> these flags are set to 1.

✕ = 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. 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.

STORAGE REPORT TABLE

Function

To hold the information from which a storage report will be generated.

When Generated

During program or task initialization.

Where Held

Program management area, or for major task in storage associated with the control task.

How Addressed

From X'38' in the TIA.

Non-multitasking and PL/I Task Table

	0	1	2	3	4	
0	End of Stack Pointer (EOS)					TRES
4	Used ISASIZE					TRUS
8	TRFG	Specified ISASIZE				TRSS
C	ISA adjustment					TRUN
10	Extra storage required					TREX
14	Number of GETMAINs					TRGM
18	Number of FREEMAINs					TRFM
1C	Number of get non-LIFO requests					TRGN
20	Number of free non-LIFO requests					TRFN
24	Current extra storage owned					TRCS
28	Current unused ISA					TRUI
2C	Address of tasking appendage (multitasking only)					TRTT
30	Heap GET Requests					TRHGN
34	Heap FREE Requests					TRHFN
38	Heap GETMAIN Count					TRHGM
3C	Heap FREEMAIN Count					TRHFM
40	Maximum Amount of Heap GETMAINed					TRHMX
44	Heap Current GETMAIN					TRHCS

FLAG BYTE (TRFG)

```
TRMT    Bit 0 = 1       Major task table
TRUC    Bit 1 = 1       Update complete (get LIFO)
        Bits 2-7        Not used
```

Control Task Table

Offset	0 — 4	Name
0	Major Task — Used ISASIZE	CSMU
4	Major Task — Specified ISASIZE	CSMI
8	Major Task — ISA Adjustment	CSMN
C	Major Task — Extra Storage Required	CSMX
10	Major Task — Number of GETMAINs	CSMG
14	Major Task — Number of FREEMAINs	CSMF
18	Major Task — Number of Get Non—LIFO Requests	CSMH
1C	Major Task — Number of Free Non—LIFO Requests	CSMJ
20	Current Extra Storage	TRCS
24	Current Unused ISA	TRUI
28	A(Tasking Appendage)	TRTT
2C	Major Task — Heap GET Requests	CMHGN
30	Major Task — Heap FREE Requests	CMHFN
34	Major Task — Heap GETMAIN Count	CMHGM
38	Major Task — Heap FREEMAIN Count	CMHFM
3C	Maximum Amount of Heap GETMAINed	CMHMX
40	Heap Current GETMAIN	CMHCS
44	Subtasks — Maximum ISASIZE Used By Any Subtask	CSXU
48	Subtasks — Minimum ISASIZE Used By Any Subtask	CSNU
4C	Subtasks — Specified ISASIZE, All Subtasks	CSSI
50	Subtasks — Maximum Storage Required, any Subtask	CSXN
54	Subtasks — Minimum Storage Required, any Subtask	CSNN

58	Subtasks — Maximum Extra Storage, Any Subtasks	CSXX
5C	Subtasks — Minimum Extra Storage, Any Subtasks	CSNX
60	Subtasks — Total Number of GETMAINs for All Subtasks	CSSG
64	Subtasks — Total Number of FREEMAINs for All Subtasks	CSSF
68	Subtasks — Total Number of Get Non—LIFO Requests for All Subtasks	CSSH
6C	Subtasks — Total Number of Free Non—LIFO Requests All Subtasks	CSSJ
70	Maximum Number of PL/I Tasks Attached	CSNA
74	Subtasks — Heap GET Requests	CSHGN
78	Subtasks — Heap FREE Requests	CSHFN
7C	Subtasks — Heap GETMAIN Count	CSHGM
80	Subtasks — Heap FREEMAIN Count	CSHFM
84	Heap Maximum Storage	CSHXX
88	Heap Minimum Storage	CSHNX

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

Passed as parameter list by compiled code.

	0	1	2	3	4	
0	A(Source or its Locator)					SSRC
4	A(Source DED)					SSDD
8	A(Target or its Locator)					STRG
C	A(Target DED)					STDD
10	SFLG	STYP	SDSA	SDFL		
14	A(FCB for File)					SFCB
18	A(Next Statement)					SRTN
1C	Save Word Used in Compiler Generated Subroutines					SAVE
20	Value of COUNT Built-In Function		Not Used			SCNT
24	Address of ONCA					SOCA
28	Area Used During GET or PUT String to Hold Dummy FCB.					SSTR

FLAG BYTE (SFLG)

SXMT	Bit 0 = 1	Transmit on input
SVDA	Bit 1 = 1	VDA used in edit-directed input
SSED	Bit 2 = 1	IBMBSED is used
SIST	Bit 3 = 1	Call to IBMSIST required after dealing with next item (Stream I/O only)
	Bits 4-7	Not used

COMPILED CODE'S DSA NUMBER (SDSA): DSA level number (used only for data-directed I/O)

TYPE CODE (STYP)

```
SCHK   X'88'  CHECK entry to data-directed I/O
SDAT   X'80'  Data-directed I/O
SLST   X'40'  List-directed I/O
SEDT   X'20'  Edit-directed I/O
STR1   X'10'  String I/O
SGET   X'04'  Input
```

DATA-DIRECTED FLAG (SDFL)

```
SDTR   Bit 0 = 1    Terminating call to data-directed output
       Bits 1-7     Not used
```

STRING LOCATOR/DESCRIPTOR

Function

Used to pass the address and the length of strings to other routines. Also for handling strings with adjustable lengths for example, 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

From an offset from register 3 known to compiled code.

```
     0         1         2         3         4
 0  |        Byte Address of String          |
    |----------------------------------------|
 4  |  Allocated Length  |F|Not Used|   F2   |
```
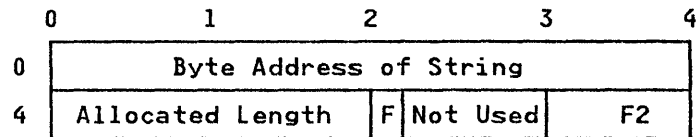
F = '0' B Fixed string (First bit of second byte)

'1' B Varying string

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.

| GRAPHIC Option of ENVIRONMENT

For the GRAPHIC option of the ENVIRONMENT attribute, the allocated length is the length that is declared. For example, if:

DCL B GRAPHIC(4)

then a value of 4 will be in the allocated length field. Length is held in number of graphics.

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 Held

Static internal control section.

How Addressed

From 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 a descriptor, which
is followed by the offset field for the next base element. If
the base element is <u>not</u> a string, array, or area the descriptor
field is omitted.

	0	1	2	3	4
0	Element Offset from the Start of the Structure			
4	Element Descriptor (if Required)			
8	Element Offset from the Start of the Structure			
C	Element Descriptor (if Required)			
	. . . For every base element in the structure, an entry is made consisting of an offset field and, if the element requires a descriptor, a descriptor.			
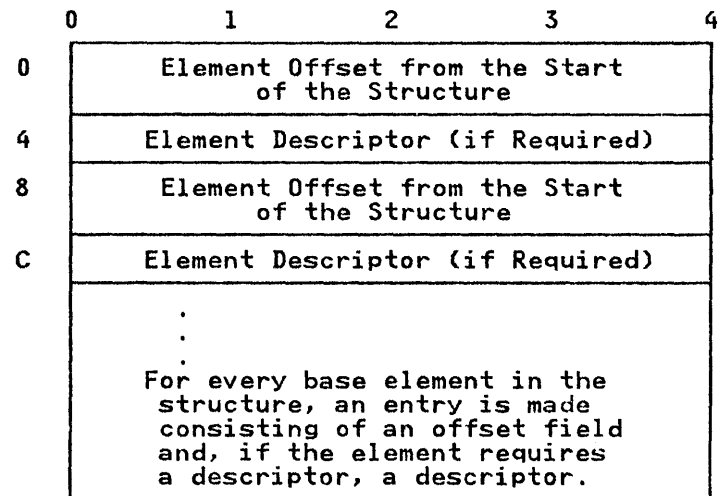
OFFSET: The offset field is held in bytes. Any adjustments
needed for bit-aligned addresses are held in the respective
descriptors.

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. External control sections consist of the name followed by an *.

How Addressed

From an offset from register 3 for internal data, by an address generated by the linkage editor for external data.

```
       0           1           2          3         4
    ┌───────────────────────┬──────────┬─────────┐
  8 │       Flags           │Dimension │ Level   │
    │  (VSF1)      (VSF2)    │ ality    │ Number  │
    ├───────────────────────┴──────────┴─────────┤
  4 │                A(DED)                       │  VSDD
    ├─────────────────────────────────────────────┤
  8 │            Address Field A                  │  VSFA
    ├─────────────────────────────────────────────┤
  C │            Address Field B                  │  VSFB
    ├──────────────────────┬──────────────────────┤
 10 │   Length of Name     │                      │
    ├──────────────────────┴──────────────────────┤
    │        Name (Fully Qualified)               │
    └─────────────────────────────────────────────┘
```
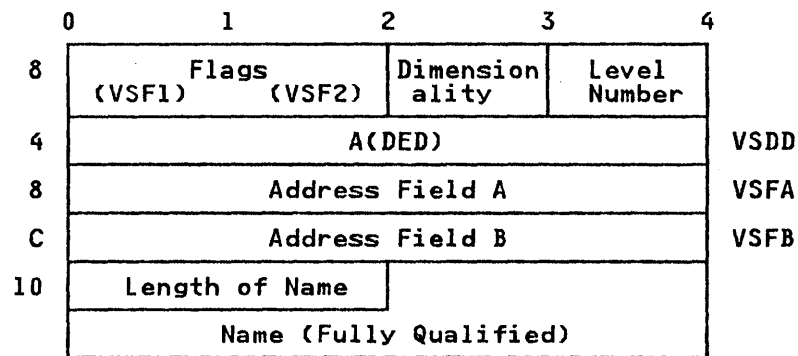
FLAGS

VSF1

VFST	X'00'	STATIC
VFPK	X'01'	Normal SYMTAB
VFSR	X'02'	A member of a structure
VFDA	X'04'	Address field A refers to data.
VFCH	X'08'	The item may appear in some CHECK list. If the item is EXTERNAL, then VFXT must also be X'10'. This field is not used if for a CONTROLLED parameter.
VFXT	X'10'	EXTERNAL
VFBS	X'20'	BASED
VFCN	X'40'	CONTROLLED (non-parameter)
VFDF	X'60'	DEFINED
VFAU	X'80'	AUTOMATIC
VFNP	X'A0'	A non-CONTROLLED parameter
VFPR	X'E0'	A CONTROLLED parameter

VSF2

VFAC	Bit 0 = 1	Address
VFTR	Bit 1 = 1	Dynamic field enabled

```
VSF2
VFDR   Bit 2 = 1      Dictionary reference precedes symbol table
VFIS   Bit 3 = 1      ISUB defined
VFB1   Bit 4 = 1      Special cases of BASED
VFB2   Bit 5 = 1      Special cases of BASED
       Bits 6 & 7     Not used
```

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 for 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 fields B and C are 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 of of the locator for items that have locators.

If CONTROLLED
      Offset of the data or its locator from the address in the anchor word.

If BASED
      Offset of field within 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: Used for CONTROLLED and BASED items only.

If CONTROLLED
      Address of anchor word, either in static internal for internal data or in a separate CSECT for external data.

If BASED
      See below.

Other data
      Not used for other data types. Set to a null value of all zeros.

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.

Where Held

Static internal control section.

```
       0         1         2         3         4
     ┌─────────────────────────────────────────┐
  0  │            A(Symbol Table)                │
     ├─────────────────────────────────────────┤
  4  │            A(Symbol Table)                │
     ├─────────────────────────────────────────┤
  8  │            Fullword of Zeros              │  <──marks end of
     ├─────────────────────────────────────────┤        block
  C  │   A(of Entries in Symbol Table Vector     │
     │   of Encompassing Block). All Zeros       │
     │        for Main Procedure Block           │
     ├─────────────────────────────────────────┤
     │                 Etc.                      │
     └─────────────────────────────────────────┘
```
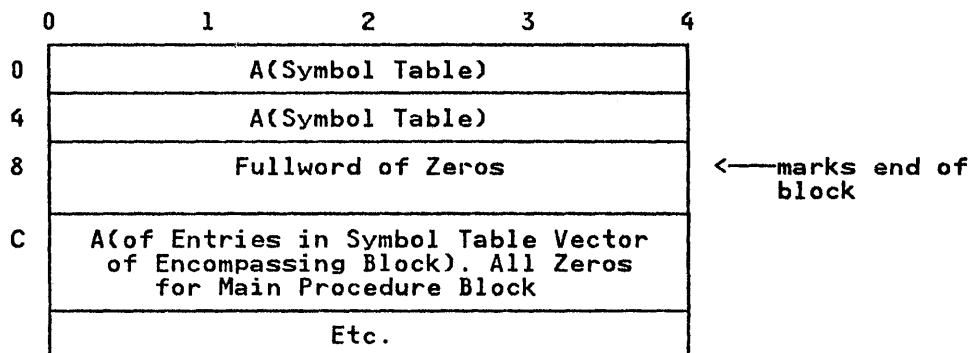
How Addressed

From an offset from register 3 known to compiled code.

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 of the entry 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.

TASK COMMUNICATION AREA (TCA)

Function

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

When Generated

During program initialization by IBMBPIR.

Where Held

In the program management area at the head of the initial segment area (ISA).

How Addressed

From Register 12

	0	1	2	3	4	
−8		"Eye Catcher"				
0	TFB0	TFB1	TFB2	TFB3		TFLG
4	A(PRV or Zero)					TPRV
8	Beginning of Segment Pointer (BOS)					TBOS
C	End of Segment Pointer (EOS)					TEOS
10	DSA next invocation count					TINC
14	A(current event variable)					TEVT
18	A(External Save Area)					TESA
1C	A(TRT Table for errors)					TTRT
20	Task Level					TTIC
24	A(Current Task Variable)					TTSK
28	A(TCA appendage)					TTIA
2C	A(Tasking Appendage)					TTTA
30	A(Save Area for Program Management)					TPSA
34	Open File Chain Anchor					TFOP
38	A(Loaded Module List)					TODL
3C	Unused					TBUG
40	A(Diagnostic File Block)					TDFB
44	PL/I Return Code (TORC)		User Return Code (TURC)			
48	A(Overflow Routine for Get VDA)					TOVV

	0	1	2	3	4	
4C	A(Flow Statement Number Table)					TSFT
50	A(Tab Table)					TTAB
54	A(Flow module)					TEFL
58	A(LPA Module — Region)					TPSR
5C	A(LPA Module — LPA)					TPSL
60	A(LPA Module — LPA)					TPSM
64	PRV Initialization Word or Zero					TPRI
68	A(Module List)					TAML
6C	A(Get Dynamic Storage Routine)					TGET
70	A(Free Dynamic Storage Routine)					TFRE
74	A(Overflow Routine for Get DSA)					TOVF
78	A(ON Condition Handler)					TERR
7C	TXAF		Not Used	TRLR	TTLR	TENV
80	Normal GOTO Code Used When GOTO Out of a Block May Occur					TGTC
F0	A(EFCL) / Dummy if No FLOW or COUNT					TEFC
F4	A(Interpretive GOTO Routine)					TGTM
F8	A(Get Control Routine)					TGCL
FC	A(Free Control Routine)					TRCL
100	A(Enqueue SYSPRINT Routine)					TEQR
104	A(Dequeue SYSPRINT Routine)					TDQR
108	A(WAIT Routine)					TAWT
10C	A(COMPLETION Pseudo-variable Routine)					TACP
110	A(EVENT Assign Routine)					TAEA
114	A(Priority Routine)					TAPR
118	A(Enqueue/Dequeue Routines)					TEDR
11C	Reserved for Users					TUSR
120	A(Attention Checking Routine)					TATP
124	A(System Dependant Appendage) Appendage under CICS (Otherwise not Used)					TCIC

Flags (TFLG)

Indicate that an abnormal GOTO out of block may take place.
Also indicate that certain special error conditions may arise.

FLAG BYTE 0 (TFB0)

TTIS	Bit 0 = 1	Subtask TCA
TTTT	Bit 1 = 1	Program may multitask
TTCK	Bit 2 = 1	Reserved for the Checkout Compiler
TTFT	Bit 3 = 1	Eldest task from attaching DSA
ITFD	Bit 4 = 1	Daughter tasks exist
TTKK	Bit 5 = 1	Operating under CICS
TTDB	Bit 6 = 1	Using a data base system
	Bit 7	Not used

Note: This flag byte is the only one in the TCA used by the
central task without synchronizing with the subtask. The
subtask must never change it. This prevents interference
between CPUs on a multiprocessing machine.

FLAG BYTE 1(TFB1)

TGFD	Bit 0 = 1	At least one daughter task may exist
TGFE	Bit 1 = 1	At least one active EVENT I/O ON-unit
	Bit 2	Not used
TGFS	Bit 3 = 1	Exit routine active SORT
TGNQ	Bit 4 = 1	SYSPRINT enqueue by this task
TGTE	Bit 5 = 1	Task ending
	Bits 6 & 7	Not used

FLAG BYTE 2 (TFB2)

THQS	Bit 0 = 1	Raise SIZE for fixed-point divide, fixed-point overflow, exponent overflow, or decimal overflow exceptions
THQI	Bit 1 = 1	Ignore fixed-point divide, fixed-point overflow or exponent overflow exceptions
	Bit 2	Not used
THCC	Bit 3 = 1	Fast/Initialization in use
THFN	Bit 4 = 1	Initialized; set to '0'B when an ON FINISH statement is executed.
THQF	Bit 5 = 1	File associated with SIZE
THOR	Bit 6 = 1	Return to caller after normal return from ON-unit
THQC	Bit 7 = 1	I/O conversion

FLAG BYTE 3 (TFB3)

TMDF	Bit 0 = 1	Dynamic FLOW set on
TPNR	Bit 1 = 1	Prompt not required
	Bit 2	Not used
TNFP	Bit 3 = 1	No floating point instructions
TNOF	Bit 4 = 1	No FLOW for this GOTO
TISN	Bit 5 = 1	Implied SKIP next
TFCT	Bit 6 = 1	COUNT required
	Bit 7	Not used

FLAG BYTE 4 (TXAF)

TX31	Bit 0 = 1	Entry AMODE(31)
TXESPI	Bit 1 = 1	ESPIE in use
TXESTA	Bit 2 = 1	ESTAE in use
TXASYS	Bit 3 = 1	Extended architecture
	Bits 4-7	Not used.

TBOS: The pointer that points the the beginning of the current
segment.

TEOS: The pointer that points to the end of the current segment. (See Chapter 6, "Storage Management" on page 84).

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

TTRT: The translate-and-test table contains code used in error handling to identify relevant ON-cells.

TPSA: This points to a preformatted DSA reserved for storage management.

TFOP: Used when closing files at the end of a job.

TORC & TURC: A Standard area to keep these codes.

TOVV: Stack overflow routine for FDAs, (see Chapter 6, "Storage Management" on page 84).

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

TTAB: The address of a table of tabulator positions used in list-directed output.

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

SHARED LIBRARY (TPSR, TPSL, & TPSM): Used when accessing PL/I library modules in the link-pack-area.

TPRI: Used to access word set in PRV when files are closed.

STORAGE POINTERS (TGET, TFRE, & TOVF): Entry points to IBMBPGRA that get non-LIFO storage, free non-LIFO storage, and acquire a new segment for LIFO STORAGE (see Chapter 6, "Storage Management" on page 84).

TERR: Address branched to after a software-detected interrupt occurs, (see Chapter 7, "Error and Condition Handling" on page 105).

TENV: Identifies release of libraries being used. See also, "Flag Byte 4 (TXAF)" on page 409.

TRLR: The resident library release number.

TTLR: The transient library release number.

TGTC: 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.

The function of this code is described under "Handling Flow of Control" on page 31.

TGCL: Routine used in multitasking, (see Chapter 14, "Multitasking" on page 307).

TEQR & TDQR: Library routines used in stream I/O (see Chapter 9, "Stream-Oriented Input/Output" on page 185).

TAWT: Address of IBMBJWT, the module used to execute the WAIT statement.

TACP: Address of COMPLETION pseudo-variable module.

TAEA: Address of event assign module.

TAPR: Address of the priority routine.

TEDR: Used for enqueuing and dequeuing files other than SYSPRINT.

TCA IMPLEMENTATION APPENDAGE (TIA)

Function

To hold control and communication information.

When Generated

During program initialization.

Where Held

Program management area. Addressed from offset X'28' in the TCA.

How Addressed

From X'28' in the TCA.

	0	1	2	3	4	
-8		"Eye Catcher"				
0		A(Byte Beyond ISA)				TISA
4		A(Old PICA)/Fake PICA				TAPC
8		A(Interrupt Handler)				TERA
C	Interrupt Mask		Flags1	Flags2		TINM
10		WIT chain anchor				TWTW
14	Anchor for Chain of Exclusive Blocks					TEXF
1C		A(Last Free Element)				TLFE
20		A(Dump Block)				TDUB
24		A(Dummy DSA)				TDDS
28		A(Get LWS Routine)				TLWR
2C		A(Extended Float Simulator)				TASM
30	Two Words for the Name of the Extended Float Simulator					TSNM
38		A(Storage Report Information)				TASR
3C	Chain of Fetched Entry Points					TFEP
40		A(STAE Exit Routine)				TAST
44	A(Housekeeping Interrupt Routine)					TERC
48		A(First Count Table)				TCTF
4C		A(Last Count Table Used)				TCTL
50	Saved A(TCA) for the Error Handler					TATC

```
         0         1         2         3         4
      ┌─────────────────────────────────────────────┐
 54   │             A(STAE Block)                   │   TABD
      ├─────────────────────────────────────────────┤
 58   │        A(PLISTART Parameter List)           │   TRPS
      ├──────────┬────────────────────┬─────────────┤
 5C   │  Flags3  │     Not Used       │  Caller's   │
      │          │                    │  Program    │
      │          │                    │  Mask       │
      ├──────────┴────────────────────┴─────────────┤
 60   │          Real EOS (LIFO Stack)              │   TXRES
      ├─────────────────────────────────────────────┤
 64   │          ISA Increment Amount               │   TXIIC
      ├─────────────────────────────────────────────┤
 68   │         Heap Initial Allocation             │   TXHIN
      ├─────────────────────────────────────────────┤
 6C   │         Heap Increment Amount               │   TXHIC
      ├─────────────────────────────────────────────┤
 70   │          Heap Initial Address               │   TXHAD
      ├─────────────────────────────────────────────┤
 74   │          Heap Storage Chain                 │   TXBOC
      ├─────────────────────────────────────────────┤
 78   │           Heap Free Chain                   │   TXLFE
      ├─────────────────────────────────────────────┤
 7C   │            Error Counter                    │   TERN
      └─────────────────────────────────────────────┘
```

FLAGS1 (TFL1)

TFLA	Bit 0 = 1	Task terminated normally
TFLS	Bit 1 = 1	SYSPRINT open STREAM print
TFLJ	Bit 2 = 1	STAE exit in progress
TFLK	Bit 3 = 1	Dump I/O in progress
	Bits 4-7	Not used.

FLAGS2 (TFL2)

TFLD	Bit 0 = 1	Caller provided ISA
TFLR	Bit 1 = 1	Storage report required
TFLT	Bit 2 = 1	STAE required
TFLP	Bit 3 = 1	SPIE required
TFLX.	Bit 4 = 1	Syntax error in program management options
TFLM	Bit 5 = 1	Multiple STAE required
	Bits 6 & 7	Not used

FLAGS3 (TFL3)

TXHFR	Bit 0 = 1	Free heap segments
TXHBL	Bit 1 = 1	Heap below the 16M line
TXIFR	Bit 2 = 1	Free ISA segments
TXHIT	Bit 3 = 1	Heap initialized
TXHPA	Bit 4 = 1	Heap preallocated
	Bit 5 & 6	Not used
TXNHP	Bit 7	No heap processing

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.

TAPC: Used to restore SPIE to that which existed when the PL/I program was called.

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

INTERRUPT MASK AND FLAGS (TINM): Wait information table (WIT) chain header (TWTW):

> Start of the chain indication which events are being waited-on in the task.

TEXF: Used when handling exclusive files.

TLFE: Address of last free area of non-LIFO storage on the free area chain. It is used as a starting point when searching the chain.

TDUB: Used when a PLIDUMP is being executed.

TDDS: Used, when abnormally terminating the program, to restore IBMBPIR's registers. This allows IBMBPIR to be reached should the DSA chain be overwritten.

TLWR: This is part of the resident library module IBMBPIR and is used to get a new allocation of library workspace and in ONCA. This routine is called after interrupts and during program initialization, (see Chapter 3, "The PL/I Libraries" on page 53).

TSAM: Used on machines that do not have the extended floating-point instructions to handle extended floating-point data.

TSNM: Used to hold the name of the extended float simulator, so that it can be invoked if required.

TCA TASKING APPENDAGE (TTA)

Function

> To hold control and communication information used in multitasking programs.

When Generated

> During program initialization.

Where Held

> Program Management area.

How Addressed

> From X'2C' in the TCA.

```
         0         1         2         3         4
      ┌─────────────────────────────────────────────┐
  0   │          POST Event Control Block           │  TPEC
      ├─────────────────────────────────────────────┤
  4   │       Parameter list for Control Task       │  TCTP
      │                 (2 words)                    │
      ├─────────────────────────────────────────────┤
  8   │          WAIT Event Control Block           │  TWEC
      ├─────────────────────────────────────────────┤
 10   │                  A(TCB)                      │  TTCB
      ├─────────────────────────────────────────────┤
 14   │             A(ECBLIST Element)              │  TAEE
      ├─────────────────────────────────────────────┤
 18   │                  A(TCA)                      │  TTCA
      ├─────────────────────────────────────────────┤
 1C   │                 Not Used                     │
      ├─────────────────────────────────────────────┤
 20   │      Chain of Sister Tasking Appendages     │  TSIS
      ├─────────────────────────────────────────────┤
 24   │       Anchor for Subtask Sister Chain       │  TSUB
      ├─────────────────────────────────────────────┤
 28   │          Anchor for I/O EVENT Chain         │  TIOE
      ├─────────────────────────────────────────────┤
 2C   │              A(Attaching DSA)               │  TDSA
      ├─────────────────────────────────────────────┤
 30   │           A(Task Invocation Point)          │  TALR
      └─────────────────────────────────────────────┘
```

Post Codes to Control Task

X'0'	Completion pseudo-variable
X'4'	EVENT assignment
X'8'	PRIORITY pseudo-variable
X'C'	I/O EVENT completion
X'10'	WAIT termination
X'14'	Detach this block
X'18'	Dedicate control task
X'1C'	Liberate control task
X'20'	Attach a task
X'24'	End of task
X'28'	Terminate a subtask
X'2C'	Terminate a subtask

TASK VARIABLE (TV)

Function

To hold information about task

When Generated

Depends on storage class

Where Held

Depends on storage class

How Addressed

From offset X'24' in the TCA.

	0	1	2	3	4
0	KFL0	KFL1	Priority		
4	A(SYMTAB)				
8	A(TCA Tasking Appendages)				
C	A(Calling PROCEDURE)				
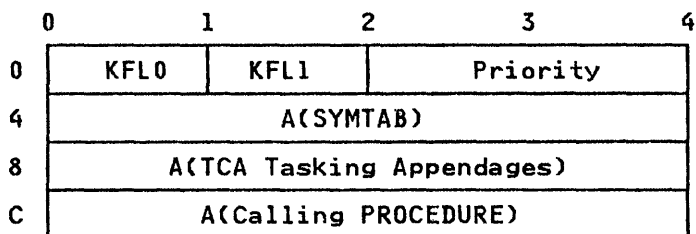
FLAGS

KFL0
KACT	Bit 0 = 1	Active
	Bits 1-7	Not used

KFL1
KDUM	Bit 0 = 1	Dummy
KSTE	Bit 1 = 1	Symbol table exists
	Bits 2-7	Not used

WAIT INFORMATION TABLE (WIT)

Function

Used to hold information about a WAIT statement

When Generated

When the WAIT statement is initiated

Where Held

In the LIFO stack

How Addressed

From X'10' in the TIA.

	0	1	2	3	4	
0	A(Chain Back)					WCHB
4	A(Events and Event Control Blocks)					WAET
8	A(Byte Beyond Table)					WABT
C	Not Used					

ZYGOLINGUAL 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 is principal procedure, at the head of the unused portion of the region immediately before the TCA.

How Addressed

From offset X'4' in IBMBILC1.

	0	1	2	3	4	
0	"Eye Catcher"					ZCTLE
4	A(Latest Interlanguage VDA) or Zero					ZCVD
8	Flag byte	Temporary Program Mask				ZTPGMASK
C	ESPIE Return for PL/I					ZCAPRET
10	ESPIE Return for COBOL					ZCACRET
14	ESPIE Return for FORTRAN					ZCAFRET
18	PL/I PICA / ESPIE Parameter List					ZCAP
1C	COBOL PICA / ESPIE Parameter List					ZCAC
20	FORTRAN PICA / ESPIE Parameter List					ZCAF
24	Work area[1] for Passing Interrupts to Entry Point IE002 in Module IBMBIEC.					ZCCP
34	Work area[1] for Passing Interrupts to Entry Point IE013 in Module IBMBIEF.					ZCFP
44	TCA Flags Save Area					ZCTF
48	A(TCA)					ZCCR
4C	Multitasking Work Area					ZCTA
50	18 Word Save Area that is Used For FORTRAN PIR and PL/I STOP					ZCS1
98	A(LWS)					ZLWS
9C	A(NAB)					ZNAB
A0		GOTO code				ZFXN
A4	STAE					ZCAE

	0	1	2	3	4	
A8	Ghost Save Area (4 words)					ZCGS
B8	Save Area for PL/I PIR (18 words)					ZCS2
100	Register Save Area for the caller of initial IBMBIEP					ZCS3
148	ESPIE Test data for PLI					ZPLI
158	ESPIE Test data for COBOL					ZCOBOL
168	ESPIE Test data for FORTRAN					ZFTN

[1]This data area is for "INTER PICA" or for an ESPIE parameter list, which is used if the INTER option is specified.

FLAG BYTE (ZCRP)

IEUVC	Bit 0 = 1	If there is a previous call to COBOL
IEUVF	Bit 1 = 1	If there is a previous call to FORTRAN
IEUNP	Bit 2 = 1	If the main program is not PL/I
	Bits 3-5	Not used
IEULT	Bit 6 = 1	STAEs will be issued
IEULP	Bit 7 = 1	SPIEs will be issued

A

ABEND analyzer (IBMBPES) 140
ABEND dump 261
 key areas of 268
abnormal GOTO statement
 changing CHECK enablement during 39
 code in TCA 80
 from an event I/O ON-unit 39
 in a termination routine 80
 library subroutine IBMBPGO 36
 out of SORT exit routine 39
abnormal locate return block 168
abnormal termination (multitasking) 319
access method
 record I/O 159
 stream I/O 185
acquiring the ISA 78
activating blocks 31
actual origin (AO) 66
address constants
 Q-type 28
addressing
 automatic variables 25
  allocated storage in DSA 25
  allocated storage in VDA 25
  beyond the 4K limit 27
 based variables 26
 beyond the 4K limit 27
 controlled variables
  pseudo-register vector 26
 files via DCLCB and PRV 166
 static variables 27
  beyond the 4K limit 27
 temporary variables 26
  allocated storage in DSA 26
  allocated storage in VDA 26
  beyond the 4K limit 27
addressing beyond 4K limit 27
adjustable extents, example of 70
aggregates (see also arrays and
 structures)
 address 65
 arrays of structures 30
 COBOL 303
 descriptor descriptor 69, 328
 FORTRAN 303
 how passed 65
 interlanguage arguments 303
 locator 68, 330
aggregates (see also arrays
 structures) 29
alignment in structures 303
ALL built-in function 232
allocation of dynamic storage 4, 85
AND logical operation 232
ANY built-in function 232
AO (see actual origin) 66
area
 control block 327
 descriptor 326
 locator/descriptor 68, 326
argument lists 39
 DO-LOOPS, use of 42
 in static storage 40
 passed by calling routine 39

setting up 41
arrays
 assignments 31
 boundaries 29
  array descriptor 30, 67
  array locator 67
 descriptor 69, 331
 elements, example of 64
 FORTRAN 303
 how passed 65
 implementation of 29
 interlanguage communication 282
 interleaved 30, 232
 multipliers 29
 of structures 30, 71
  interleaved 232
  locators and descriptors 71
 program control data 29
 virtual origin 29
ASSEMBLER - PL/I communication 305
ASSEMBLER option 305
attaching a task 317
ATTENTION condition 51
attention interrupt 51
attributes, data 64
automatic variables
 addressing beyond the 4K limit 27
 definition 25
 in dump 277
 initialization of 32
 storage in DSA 25
 storage in VDA 25, 85

B

backchains
 dynamic 33
 in multitasking 313
 static 33, 134
base element 66
base registers
 DSA pointer 24
 program base 23
 static base 23
 TCA pointer 23
based variables 26
 in dump 277
 storage 102
beginning-of-segment (BOS) pointer 88
BIT data
 string assignment subroutine
  (IBMBBGF) 232
block enable cells 123
blocks
 activating 31
 terminating 31
BOOL built-in function 232
BOS (beginning-of-segment) pointer 88
bounds, adjustable 64
branch-in
 followed by branch-in 148
 following a branch-out 148
 to a new block 149
 to an ON-unit 150
branch-out

following a branch-in 148
branches, rationalization of 50
buffer control fields (stream I/O) 191
buffer pointers (stream I/O) 191
built-in functions
    arithmetic 230
    array handling 231
    condition 119
    DATE 235
    library subroutines 230
    mathematical 230
    string handling 231
    structure handling 231
    TIME 235
built-in subroutines
    checkpoint/restart 239
    sort/merge 236

```
┌───┐
│ C │
└───┘
```

C format item DED 343
CALL statements 35
CALL...TASK failure 318
calling trace
    following through dump 272
    obtaining 254
chain, free area 79
CHECK condition 109, 131
    handling 133
    raising 131
    testing for enablement 132
CHECK prefix 131
checking code 117
checkpoint/restart facility 239
CHKPT macro instruction 239
CICS
    appendage 333
    error handling under 151
    execution-time options 83
    initialization/termination under 82
    modules in resident library 54
    PLIDUMP on 151
    program management under 82
    storage management
      considerations 103
    stream I/O under 219
CLOSE statement
    compiler output 174
    general 156
closing files
    explicit closing 174
    implicit closing 157, 174
    library subroutines 174
COBOL
    COBOL-PL/I communication 297
    interrupt 298
    option in ENVIRONMENT attribute 306
    structure mapping 303
    ZERODIVIDE ON-unit 298
COLUMN format item 218
common expression, elimination of 44
commoning
    aggregate descriptor descriptors 69
    array and structure descriptors 67, 69
    for optimization 50
communication
    between languages 281-306
    between routines 64
    between tasks 309

compare-aligned-bit-strings subroutine
  (IBMBBBC) 232
compare-unaligned-bit-strings subroutine
  (IBMBBGC) 232
compilation, definition 1
compile-time DED 71
compiler options
    AGGREGATE 15
    COUNT 141
    ESD 15
    FLOW 141
    LIST 15
    MAP 15
    OFFSET 15
    SOURCE 15
    STORAGE 15
compiler output 12-52
    control sections 12
    dummy sections 14
        pseudo-register vector 14
    ESD records 12
    relocatable object module 12
    RLD records 12
    TXT records 12
        constants 12
        machine instructions 12
compiler-generated subroutines 203
    IELCGIX 208
    IELCGOC 208
    purpose of 43
COMPLETION
    built-in function 240
    pseudo-variable 240
completion values, multitasking 320
computational subroutine 230
concatenate-character-strings subroutine
  (IBMBBCK) 232
CONDITION condition 136
conditions
    default values 119
    defaults 107
    enablement 106
    general 119
    implementation in general 111
    name abbreviations in dump 256
consecutive buffered files 158
constants 21
constants pool 21
contents of listing information 14
contents of load module 8
control blocks
    array descriptor 30
    for communication between routines
        aggregate descriptor
          descriptors 69
        data element descriptors 71
        descriptors 67
        locators 67
        symbol table vectors 72
        symbol tables 72
    for optimization
        commoning 50
    formats 326-418
    in a PL/I environment 3
    locating in dump 278
    non-VSAM section 376
    structure descriptor 30
    summary of uses of 113
    VSAM section 376
control format information 201
    DED 343
control sections 12-52
    PLICOUNT 14
    PLIFLOW 13

disabled condition 105
DISPLAY statement 235
DO-loops
  accessing array elements
    example of 48
    loop control variables 48
  example of 42
  nested 42
DSA (see dynamic storage area)
DST (diagnostic statement table) 395
DUB (dump block) 139, 349
dummy arguments in interlanguage
 communication 281
dummy DSA 75, 81
dummy FCB 29
dummy ONCA
  chaining 119
  definition 79
  description 75
  format 381
dummy sections
  constants 12
  machine instructions 12
  pseudo-register vector 14
dump block (DUB) 139, 349
dump control module (IBMBKMR) 137
dump debugging procedures 258-262
dump file (PLIDUMP) 139
dump module output 139
dump routines
  interrelationship of 138
dumps
  contents of 254
  debugging with 248-280
  file information in 256
  hexadecimal 257
  housekeeping information 269
  implementation 137
  locating information in
    ABEND dump 268
    control blocks and fields 278
    finding variables 277
    list of contents 263
    PL/I dump 264
    stand-alone dump 269
  obtaining 250, 251
  options 251
  subroutines that generate 137
  system ABEND 261
  trace information in 254
  use of block option 258
dynamic backchain 33
dynamic descendency 106
dynamic ONCB 383
dynamic storage allocation 4, 85
dynamic storage area (DSA)
  affecting PL/I environment 75
  associating DSA with block 272
  automatic variables 4
  chaining of 267
  chaining when multitasking 280
  contents for compiled code DSA 34
  definition 75
  dummy 75, 81
  finding main procedure DSA in
   dump 274
  following back-chain in dump 269
  format and function 346
  forward chain in dump 274
  housekeeping information 4
  IBMBERR's DSA in dump 272
  LIFO storage stack 4, 6, 85
    segment handling 97
  major free area 85

non-LIFO storage stack 85, 91
  library routine IBMBPRG 6
  pointers used in allocation
    beginning-of-segment (BOS) 88
    byte beyond the ISA (TISA) 89
    end-of-segment (EOS) 88, 91, 96
    free area chain (TLFE) 88
    next-available-byte (NAB) 88, 96
  prolog code 4
    dummy DSA 4, 81
    initialization routine 4
  reducing storage requirements 3
  register save area in 273
  register save information 4
  transient library 53
  uses 85
  variables in
    automatic 3
    based 3
    controlled 3

| E |

E format item DED 342
ECB (event control block) list 309
edit-directed I/O 201-209
  code generated 206
  compiler-generated subroutines 203
  data format items 201
  FED 201
  format DED 201
  format list 208
  format option handling 208
  GET statement 201
  handling control format items 208
  library director modules 216
  matching data and format lists 205,
   208
  nonmatching data and format
   lists 208
  PUT statement 201
  typical statement 204
  use of library in 203
element
  base 66
  structure 66
elimination of unreachable
 statements 47
enabled condition 105
enablement status 36
end of file 180
END statement 31
end-of-segment (EOS) pointer 88
ENDFILE condition
  detection of 117
  in record I/O 180
  in stream I/O 211
  summary information 109
ENDPAGE condition 109
enqueuing on SYSPRINT 325
entry data control block 350
entry points
  conversion subroutines 222
  EXTRN 62
  for initialization/termination
    IBMBPIRA 77
    IBMBPIRB 77
    IBMBPIRC 77
  for storage management
    IBMBPGRA 96
    IBMBPGRB 96

F

special case coding, reduction
     of 6
storage report 99
stream I/O subroutines 216
under CICS 82
input/output control block (IOCB) 173,
   374
INTER option 295
interlanguage communication 281, 306
   aggregate arguments 283, 304
   arrays 303
   ASSEMBLER option 305
   basic rules 281
   COBOL calls PL/I 286, 301
   COBOL option of ENV attribute 306
   control blocks 289
   data aggregate differences 282
   FORTRAN calls PL/I 286, 301
   IBMBILC1 290
   interrupt handling 298, 300
   interrupt in COBOL 298
   interrupt in FORTRAN 300
   interrupt in PL/I 295
   NOMAP option 304
   NOMAPIN option 304
   NOMAPOUT option 304
   PL/I calls COBOL 284, 297
   PL/I calls FORTRAN 284, 299
   principles of 283
   structures 303
   tail code subroutines 287
   use of locators 282
   VDA 290
   ZCTL 290
interlanguage housekeeping routines 283
interlanguage root control block
   (IBMBILC1) 290, 371
interlanguage VDA (ZVDA) 290, 372
interleaved arrays 232
internal form of data types 221
interpretive GOTO routine IBMBPGO 36
interrupt
   floating point underflow 128
   return to the point of 130
   software 128
interrupt block 118
interrupt control block (ICB) 373
interrupt control block, definition 117
interrupt handling 105-153
   COBOL 298
   FORTRAN 300
   IBMBERR module 111
   passing information 118
   return from 130
interrupt identification using
dump 248-280
INTERRUPT option 51, 80
interrupts
   acquiring information about 111
   detection of PL/I conditions 111
   how detected 107
invariant expressions
   elimination of 46
invert-aligned-bit string subroutine
   (IBMBBBN) 232
IOCB (input/output control block) 173,
   374
ISA (see initial storage area) 75

KD (key descriptor) 167, 378
KEY condition 109
key descriptor (KD) 167, 378
key variable 167

label data control block 379
label data format 379
label variables
   errors when using 38
   format 379
   general description 38
   how passed 65
   in GOTO statements 38
      with NOOPTIMIZE 38
      with OPTIMIZE (TIME) 38
last free area (TLFE) 88
last-in/first-out (LIFO) storage
   allocation of
      diagram of principles involved 89
      low-address end of ISA 85
      prolog code 89
      segment of LIFO stack 93
   contents of 85
   freeing of
      diagram of principles involved 89
      epilog code 89
   kinds of storage areas
      dynamic (DSA) 85
      variable data (VDA) 85
   major free area 85
   segment handling
      diagram of format for 98
      entry points 97
      GETMAIN macro instruction 98
      special save area in TCA 97
      storage for DSA 97
      storage for VDA 97
      storage management routine 95
      value of EOS pointer 97
      value of NAB pointer 97
library
   resident 53
   shared 58
   transient 53
   workspace
      allocation of 56, 75
      diagram of 55
      format of 56
library calls
   addressing a subroutine 41
   example of 40
   general 40
   level of optimizing used 40
   mnemonic letter usage 41
   naming conventions 40
   resident library
      bootstrap routines 40
   setting up argument lists 41
   within TCA 42
library modules
   transient 139
library register usage 24
library routine
   IBMBSPL 208
   IBMBSXC 208

library subroutines
   arithmetic 230
   array handling 231
   computational 230
   conversion package 221
   IBMBAIH (interleaved
    array-handling) 232
   in record I/O 157
   in stream I/O 218
   mathematical 230
   naming conventions 54
   PL/I libraries 6
      OS PL/I Resident Library 6
      OS PL/I Transient Library 6
   special-case coding, reduction of 6
   string handling 231
   workspace 55
library workspace (LWS)
   definition 75
   format and function 380
library-call I/O 156, 167
   implicit open 176
LIFO storage (see last-in/first-out
 storage)
LINE format option 208
link-editing
   for fast-path
    initialization/termination 76
   in a load module 7
      error handler, IBMBERR 7, 74
      initialization routine,
       IBMBPIR 7, 74
      OS data management routines 7
      resident library routines 7
link-editing, definition 3
link-pack-area (LPA)
   communication with program region 59
   during initialization 62
   transfer vector 58
list-directed I/O 193
listing conventions 14, 18
load module
   contents 7
      compiler output 7, 12
      link-edited routines 7
      of typical 7
      PLIMAIN 7
      PLISTART 7
   entry point 12
LOCATE statement 154, 168
locators 282
   aggregate 68
   aggregate locator format and
    function 330
   area 68
   area locator/descriptor format and
    function 326
   commoning 67
   location of 67
   string 68
   string locator/descriptor format and
    function 402
   variables, how held 64
logical operation subroutines 230
loops
   rationalization of program
    branches 50
LWS (library workspace)
   definition 75
   format and function 380

M

major free area
   allocation of LIFO storage
      diagram of principles involved 89
   diagram of elements on free area
    chain 97
   location in DSA 85
modification of DO-loop control 48
module, object 12
movement of expressions out of loop 46
multiple conversion 228
multitasking 307, 325
   acquiring the ISA during 103
   back-chains in 313
   completion values 320
   following chaining in dump 275
   housekeeping 311
   library 315
   modules in multitasking library 312
   POSTCODEs 310
   priority 311
   reading dumps, general 280
   shared library considerations 62
   storage reports 101
   TCA tasking appendage (TTA) 309
   wait module IBMTJWT 323
   WAIT statement 320

N

NAB (next-available-byte) pointer 88
NAME condition 109
   in stream I/O 211
naming conventions
   of conversion modules 222
   of library modules 54
next-available-byte (NAB) pointer 88
NOMAP option 304
NOMAPIN option 304
NOMAPOUT option 304
non-LIFO storage
   allocation of
      diagram of principles
       involved 91, 92
      end-of-segment pointer (EOS) 91
      error during 96
      GETMAIN macro instruction 91, 96
      high-address end of ISA 85
      library module IBMBPGR, in TCA 91
   contents
      based variables 85
      controlled variables 85
   freeing of
      FREEMAIN macro instruction 91, 96
      TISA field of ISA 96
   major free area
      diagram of free-area chain
       element 97
      free-area chain 87, 91
non-VSAM section of control blocks
   data management event control
    block 376
normal return 106
NOSPIE option 80
NOSTAE option 80
null ON-unit 125
null value 29

PLIDUMP facility
   implementation 137
PLIFLOW 13, 146
PLIMAIN 74
   format 387
   in a load module 7
PLISHRE 58
PLISORT 236, 238
PLISTART
   for use in link-editing 74
   in a load module 7, 74
   parameter list 388
PLITABS 210
PLIXOPT 77, 82
pointer variables
   misuse of 259
pointers
   COPY option 212
   how held 25
   null value 29
   storage handling 88
   used in dynamic storage allocation
      beginning-of-segment (BOS) 88
      byte beyond the ISA (TISA) 89
      end-of-segment (EOS) 88, 91, 96
      free area chain (TLFE) 88
      next-available-byte (NAB) 88, 96
      real end-of-segment (TXRES) 88
POLY built-in function 232
POST ECB 310
POSTCODEs, list of 310
print files, formatting for 210
priority of task 311
privileged operation interrupt 106
procedure block
   epilog code 32, 80
   prolog code 32
PROCESS statement
   LIST option 18
program base register 23
program check exit 261
program check interrupt
   definition 105
   listing 105
program control data
   data aggregates 29
      arrays 29
      structures 29
program control section
   general registers 23
      dedicated 23
      work 23
program initialization 75
program listing information 14
program management area
   diagram of 79
   dummy DSA 7, 81
   initial storage area (ISA) 7, 75, 78
   initialization 78
   library workspace 55, 75, 81
   PRV, location of 29
   task communication area (TCA) 4, 7,
   75
program management under CICS 82
prolog code
   acquiring a dynamic storage area
   (DSA) 32
   allocating LIFO storage 89
   example of 32
prompting 214
protection interrupt 106
pseudo-register vector (PRV)
   addressing controlled variables 27,
   82

addressing files 27, 82
   file control blocks (FCBs) 28
ALLOCATE statement 27
initialization of 29
location of 29
Q-type address constants 28
use of 28
purge task subroutine 319
PUT statement 193

$$\boxed{Q}$$

qualified condition 105

$$\boxed{R}$$

RCB (request control block) 167, 391
RD (record descriptor) 167, 390
READ statement 154
real end-of-segment pointer (TXRES) 88
recompilation to obtain dump,
 avoiding 253
RECORD condition 109
record descriptor (RD) 167, 390
record I/O
   control blocks generated 160
   definition 154
   error handling
      fields used in 179
   error modules 178
   FCB 166
   implementation 154
      access method 159
      CLOSE statement 156
      fields used for 160
      file declaration 154, 161
      implicit close 157
      implicit open 156
      OPEN statement 156, 162
      transmission statement 156
   in-line 180
   interface routine (IBMBRIO) 167
   library subroutines used 157
   overview 183
   VSAM data sets 163
reducing storage requirements
   dynamic storage allocation 3
redundant expressions
   branching around 49
   example of 49
REFER option 234
register usage
   in dumps 273
   normal 275
registers 22-25
relative virtual origin (RVO)
   definition of 66
   of arrays 69
RELEASE statements
   DELETE macro instruction 52
REPEAT built-in function 232
REPLY option 235
report table 100
request control block (RCB) 167, 391
   description 161
resident library 53
   control name 54
   link-edit name 54

T

tab table 210
tail code subroutines 287
task communications area (TCA) 23
    description 75
    flags 80, 409
    format and function 407
    implementation appendage 87, 411
    in PL/I environment 4, 75
    special save area 97
    tasking appendage (TTA) 275, 309
task variable
    how passed 65
task variable (TV)
    format and function 415
task, finding relationship between in dumps 275
TASKHEAP option 84
tasking appendage (TTA) 275, 309
TCA (see task communications area)
temporaries (see temporary variables) 26
temporary variables
    addressing beyond the 4K limit 27
    storage in DSA 26, 85
    storage in VDA 26
terminating blocks 31
termination
    code 81
    fast-path 76
    process of
        abnormal-GOTO-subroutine 80
        epilog code 80
        GOTO-out-of-block 80
        SPIE macro instruction 80
        STAE macro instruction 80
        TCA flags 80
    storage report 100
    under CICS 82
termination of program 80
TIA (TCA implementation appendage) 411
TIME built-in function 235
timestamp 268
TLFE (last free area) 88
TOTAL option 156
trace, information in dump
transfer vector 58
transient library 53
transient library modules 139
transient open routines 163
TRANSLATE built-in function 232
transmission statements
    compiler output 167
    definition 154
    ENDFILE routine 180
    error conditions in 176
    execution of 170
    general error routines 180
    in record I/O 156
    TRANSMIT condition 180
    use of EVENT option 171
TRANSMIT condition 109, 180
    detection of 117
    in stream I/O 210
transmitter interface module (IBMBRIO) 167
transmitter modules
    record I/O 157
    stream I/O 214
TTA (TCA tasking appendage)
    function and format 413

general 309
TV (task variable)
    format and function 415
TXRES (real end-of-segment) pointer 88
TXT records 12

U

unaligned bit strings 221
UNDEFINEDFILE condition 109
UNDERFLOW condition 108
unexpected end of file
    in stream I/O 211
UNLOCK statement 154
unqualified condition 105
unreachable statements
    elimination of 47
use of locators 282
use of storage 9

V

variable data area (VDA)
    description 25
    interlanguage communication 290
variables
    automatic 25
    based 26
    controlled 26
    EXTERNAL 74
    label 379
    locating in dump 277, 280
    pointer 26
    static 27
    temporaries 26
variables, handling and addressing 25
VDA (see variable data area)
vector, symbol table 72
VERIFY built-in function 232
virtual origin (VO) 66
VO (virtual origin) 66
VSAM data sets
    opening 163
VSAM section of control blocks
    data management event control
        block 376

W

WAIT ECB 310
wait information table (WIT) 321, 416
WAIT statement 240
    example of implementation
        problems 242
    label variable
        example of 38
        with NOOPTIMIZE 38
        with OPTIMIZE (TIME) 38
    multitasking 320
    nonmultitasking 241
    summary of 246
    termination of 39
    use of event tables 242
weak external reference (WXTRN) 57
WIT (wait information table) 321, 416

work registers 24
    floating point registers 24
    library registers 24
WRITE statement 154
WXTRN (weak external reference) 57

X

X format item 218
XBF (exclusive block file) 358
XBI (exclusive block IOCB) 356

Z

ZCTL (zygolingual control list) 290,
  417
ZERODIVIDE condition 108
ZERODIVIDE ON-unit 298
ZVDA (interlanguage VDA) 290, 372
zygolingual control list (ZCTL) 290,
  417

OS PL/I Optimizing Compiler:
Execution Logic
SC33-0025-3

This manual is part of a library that serves as a reference source for systems analysts, programmers, and operators of IBM systems. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate.

Note: *Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.*

Note: Staples can cause problems with automated mail sorting equipment. Please use pressure sensitive or other gummed tape to seal this form.

List TNLs here:

If you have applied any technical newsletters (TNLs) to this book, please list them here:

Last TNL _____

Previous TNL _____

Previous TNL _____

Fold on two lines, tape, and mail. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail directly to the address in the Edition Notice on the back of the title page.) Thank you for your cooperation.

SC33-0025-3

Reader's Comment Form

|||| |||

BUSINESS REPLY MAIL
FIRST CLASS   PERMIT NO. 40   ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
P.O. Box 50020
Programming Publishing
San Jose, California 95150

IBM ®

OS PL/I Optimizing Compiler Execution Logic (S370-29) Printed in U.S.A. SC33-0025-3