File No. S360-29
Form Y28-6801-4

Program Logic

IBM System/360 Operating System
PL/I Subroutine Library
Program Logic Manual

Program Number 360S-LM-512

This publication describes the internal specifications of the PL/I Subroutine Library as a system component of IBM System/360 Operating System. The relationships between the code produced by the PL/I (F) Compiler, the PL/I Library modules and the control program are described, and summaries of the properties of individual modules are provided. This information is intended for use by those involved in program maintenance and by system programmers who are altering the program design. Program logic information is not necessary for the use and operation of the program; therefore, distribution of this publication is limited to those described above.

Restricted Distribution

Fifth Edition (November, 1968)

This is a major revision of, and obsoletes, Y28-6801-3 and Technical Newsletter Y33-6001. A new section has been added in Chapter 5 on multiprocessing and in Chapter 6 on interrupt handling for the System/360 Model 91. Other changes to the text, and small changes to illustrations, are indicated by a vertical line to the left of the change; changed or added illustrations are denoted by the • symbol to the left of the caption.

This edition applies to Release 17, of IBM System/360 Operating System, and to all subsequent releases until otherwise indicated in new editions or Technical Newsletters. Changes are continually made to the specification herein; any such changes will be reported in subsequent revisions or Technical Newsletters.

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

Address comments concerning the contents of this publication to IBM United Kingdom Laboratories Ltd., Programming Publications, Hursley Park, Winchester, Hampshire, England.

PREFACE

This publication describes the object-time PL/I Library package which forms an integral part of the PL/I processing system. General information covering the overall design and conventions is provided as well as information specific to the various areas of language support.

The publication is intended primarily for technical personnel who wish to understand the structure of the library in order to maintain, modify, or expand the PL/I processing system.

Information relevant to this manual is contained in the following IBM publications:

IBM System/360: Principles of Operation, Form A22-6821

IBM System/360: PL/I Reference Manual, Form C28-8201

IBM System/360 Operating System:

Assembler Language, Form C28-6514

Introduction, Form C28-6534

Concepts and Facilities, Form C28-6535

Linkage Editor, Form C28-6538

Job Control Language, Form C28-6539

System Programmer's Guide, Form C28-6550

System Generation, Form C28-6554

PL/I Subroutine Library Computational Subroutines, Form C28-6590

PL/I (F) Programmer's Guide, Form C28-6594

System Control Blocks, Form C28-6628

Supervisor and Data Management Services, Form C28-6646

Supervisor and Data Management Macro Instructions, Form C28-6647

PL/I(F) Compiler, Program Logic Manual, Form Y28-6800

PL/I Language Specifications, Form Y33-6003

The publication includes two introductory chapters, 'The PL/I Library' and 'General Implementation Features', which contain a general description of the library as a component of IBM System/360 Operating System, and general notes on features of the operating system and the PL/I (F) Compiler that are used in the library implementation. The remainder of the manual describes the design of the library modules in relationship to PL/I language features, and indicates the use that is made of the control program to support the design.

The descriptive material is supported by a set of module description summaries and several appendixes. The module summaries indicate the salient features of individual modules in the library package, and act as guides to the program listings that are available as part of the PL/I Library distribution. The appendixes contain details of the system macro instructions used, system generation, library pseudo-registers and macro instructions, library internal error codes and associated messages, and PL/I control blocks.

FIGURES

FUNCTION

The PL/I Library is a set of object-time modules that, in various combinations, supplement compiler-generated modules to produce executable programs.

The library modules can be divided into two groups:

1. Modules that serve as an interface between compiled code and the facilities of the control program of IBM System/360 Operating System. The main areas concerned here are input/output, dynamic program and storage management, and error and interrupt handling.

2. Closed subroutines designed to perform the data processing operations required during program execution. The areas concerned here are I/O editing, data conversion, and the computational operations necessary for the implementation of the arithmetic, floating-point arithmetic, array and string generic built-in functions.

User-designed modules can be substituted for library modules; each user module is given the name of the library module it is meant to replace.

CHARACTERISTICS

The PL/I Library was designed as a large number of modules to ensure that the object program would contain only functional code, and to simplify maintenance and modification of the library. Each module is intended to perform a single recognizable function or a group of related functions and is used alone or in combination with other library modules.

All library modules are designed for use in a multiprogramming or multitasking environment. They are therefore reenterable; they can be used by more than one task at a time, and a task may begin executing a module before a previous task has finished executing it.

The library modules are reenterable because neither the instruction code nor the data areas in them are modified during execution. The PL/I program in which they are used is protected against accidental modification by another program during execution by a protection key provided by the control program.

USAGE

The linkage editor combines the compiled modules with the library modules they require, using the external symbol dictionary (ESD). The ESD resolves all direct references to the library modules; these references can be to module names (containing five or six characters) or to entry-point names (containing seven characters). (See 'Naming Conventions' in Chapter 2 for definitions of these names.)

However, the library modules may in turn call other library modules (as, for example, in data conversion). To call these secondary modules and to ensure that only the ones required are called, a technique of non-obligatory symbol resolution is used. Any library object module that calls a secondary module that may only occasionally be required is preceded by a linkage editor LIBRARY statement that specifies that the references to the secondary modules (which are in the form of seven-character entry-point names) should not be resolved unless the modules are already part of the input to the load module. For those secondary modules that are required, the compiler generates another ESD, in which the references to the modules are in the form of five-character or six-character module names. These references can now be resolved, and the required modules are placed in the input stream.

The PL/I Library for each version of the F Compiler is compatible with previous versions. For example, a module compiled under Version 2 can be link-edited and executed by an operating system that includes the Version 3 compiler. But a module compiled under any version of the compiler cannot be link-edited by an operating system that uses an earlier version.

Compatibility is discussed fully in IBM System/360 Operating System: PL/I (F) Programmer's Guide.

CONTROL PROGRAM INTERFACES

The PL/I Library is the sole interface between object code produced by the PL/I compiler and the operating system. No supervisor call (SVC) or system macro instruction is issued by the compiled code produced by a PL/I compiler; a library call is made instead. This scheme safeguards compiled programs and the compilers from changes in operating system specification. When the operating system changes, only the library module is rewritten; the call to the library from the compiler remains as before.

System macros and SVCs are necessary because certain facilities, such as input/output functions and the timer, can only be used through the services of the supervisor. The supervisor must control these facilities so that it can preserve the integrity of its own lists, tables and control blocks, which, in a multiprogramming system, may be associated with many tasks, programs or jobs. The operating system requires that certain operations be carried out only in supervisor mode; it is an error if such instructions are executed in problem program mode.

Although it might be possible in some instances to issue SVC instructions directly to the supervisor, the use of system macro instructions is more convenient. These system macros bear a similar relationship to the supervisor and assembly code as the library module does to the operating system and the compiler. If the SVC calling sequence changes, the macro is changed to fit, and the program need only be reassembled.

Macro instructions are processed by the assembler program, and their expansions are in-line. The expansions contain the calling sequence together with either an SVC instruction or a branch to a control program routine. Parameters are passed either in registers or in data areas. If they are passed in registers, registers 0 and 1 are used. If they are passed in data areas, then register 1 will contain the address of the data area; this register is called the parameter list register.

The two main types of macros, therefore, are:

1. R-type, where parameters are passed in registers.

2. S-type, where parameters are passed in a data area.

The macro instructions used by the library are listed in Appendix A.

For further details about macro instructions, see IBM System/360 Operating System: Supervisor and Data Management Macro Instructions.

OPERATING SYSTEM REQUIREMENTS

Diagnostic File

During execution time, it may be necessary to inform the user of various error conditions as they arise. To achieve this, the job step in which the program is being executed requires a DD statement for a diagnostic file. The ddname for the file is SYSPRINT. In the absence of this statement, any diagnostic messages that arise will be printed on the system console.

Link Library

Certain modules are loaded dynamically during program execution. These modules reside in the link library (SYS1.LINKLIB); they are transient modules and are loaded, when required, by the system macros LINK, LOAD and XCTL. DD statements are not required. The link-library modules are marked * in Appendix G; they comprise:

1. The print and message modules of the error and interrupt handling subroutines.

2. The modules for opening and closing files.

3. The record-oriented I/O transmission modules.

These modules can, if required, be replaced by user-designed modules. The user module is loaded by the linkage editor into a partitioned data set (PDS). The PDS (which may have to be provided for the purpose) must appear in a JOBLIB DD statement.

INSTRUCTION SET REQUIREMENTS

The universal instruction set is generally required for the execution of PL/I programs. It is possible that floating-point or decimal instructions may be used in the execution of programs that do not use floating-point or decimal data.

NAMING CONVENTIONS

The PL/I Library conforms to the naming conventions of IBM System/360 Operating System with regard to external names. These are names that are identifiable outside the bounds of an assembled or compiled module. PL/I external names always begin with IHE; this is followed by two, three or four characters, according to the name function (see Figure 1).

REGISTERS: SYMBOLIC NAMES

The following symbolic names are used in the library modules for general registers 0-15:

Register	Symbolic Name	Register	Symbolic Name
0	R0	8	RH
1	R1,RA	9	RI
2	RB	10	RJ
3	RC	11	RX,WR
4	RD	12	PR
5	RE	13	DR
6	RF	14	LR,RY
7	RG	15	BR,RZ

The following symbolic names are used for the floating-point registers:

Register	Symbolic Name
0	FA
2	FB
4	FC
6	FD

LINKAGE CONVENTIONS

Linkage between modules generally follows the operating system standard calling sequence. The main features of this are:

1. Arguments are passed by name, not by value. The addresses of the arguments are passed, not the arguments themselves.

2. These addresses are stored in a parameter list.

3. The address of the list is stored in register RA.

Full details are provided in IBM System/360 Operating System: Supervisor and Data Management Services.

Some PL/I Library modules, however, are called by a PL/I standard calling sequence. The main features of this are:

1. Arguments are passed by name.

2. Arguments are passed in general registers.

Number of Characters	Format	Use	Meaning
5	IHEXX	Module name	XXX are chosen for mnemonic identification of function.
6	IHEXXX	Module name	XXX are chosen for mnemonic identification of function.
6	IHEXXX	PL/I Library defined macros	XXX are chosen for mnemonic identification of function.
7	IHEXXXX	Entry-point name	First six characters are module name; the seventh identifies the entry point within the module.
7	IHEQXXX	Pseudo-register name	XXX are chosen for mnemonic identification of function. (See Appendix C.)

Figure 1. External Names used by the PL/I Library

This standard can only be used where the number of arguments is both fixed and less than eight. If these conditions are not met, the operating system standard is used.

Two PL/I Library modules, IHESA and IHETSA, do not use either of these standards. The subroutines in these modules pass arguments by value as well as by name, and pass them in parameter lists and in general registers.

Whichever standard is used, whenever one module links to another a save area must be provided for the contents of the registers used by the called module. The save area procedure is:

1. The calling module provides a standard save area (SSA) for the called module. The address of this save area is stored in register DR.

2. If the called module in turn calls another module, it provides that module with a save area. Register DR now contains the address of this new save area. The save areas are chained together by the chain-back address field in the new save area.

3. On return to the calling module, the following will be <u>unchanged:</u>

   Registers RB through LR

   Program mask

   Program interrupt control area (PICA)

   while the following <u>may be changed:</u>

   Registers R0, RA, and BR

   Floating-point registers

   Condition code

The standard save area is a 72-byte area in which the contents of all the general registers can be saved. The format is described in Appendix H.

The library does not support inter-module trace. Therefore:

1. The chain-forward field in the SSA is not set.

2. Calling sequence and entry-point identifiers are not employed.

CODING CONVENTIONS

Because all modules within the PL/I Library are coded to be reenterable, the following coding constraints must be observed:

1. The modules are read-only.

2. Workspace (for save areas and temporary work areas) is obtained within an area dynamically allocated at program initialization or by a call to the Get VDA (variable data area) subroutine in IHESA. (See 'Library Workspace' in this chapter and in Chapter 4.)

LIBRARY MACRO INSTRUCTIONS

Seven macro instructions are available for use in the library modules; they reside in SYS1.MACLIB. Five of these, IHEEVT, IHELIB, IHEXLV, IHEZAP, and IHEZZZ, set up symbolic definitions in the program listing and the other two, IHESDR and IHEPRV, set the current addresses of the standard save area and the pseudo-register vector (PRV) respectively. The library macros are described in Appendix D.

DATA REPRESENTATION

Three types of data may exist within a PL/I program:

1. Arithmetic

2. String

3. Statement-label

The internal representation and other details of these three types are shown in Figures 2, 3, and 4. The invocation count used in the statement-label data representation is described in Chapter 4.

Arithmetic or string data may be specified with the PICTURE attribute. A PICTURE arithmetic data item is called a <u>numeric field</u> and is represented internally as a character string. An arithmetic data item without a PICTURE attribute is called a <u>coded arithmetic data item</u> (CAD) and is represented internally in one of three System/360 formats:

Fixed-point binary
Floating-point
Packed decimal

Data Type		Implementation			
Scale	Base	Precision	Internal format	Alignment	Processing
REAL data					
Fixed	Binary	p,q Max p: 31	Fixed-point binary	Word	Arithmetic operations are performed on p-digit integers: scale factor q is specified in a DED. (See Appendix H, 'Data Element Descriptor'.)
Fixed	Decimal	p,q Max p: 15 (see note)	Packed decimal	Byte	The p digits occupy FLOOR ((p + 2)/2) bytes. Arithmetic operations as for fixed binary
Float	Binary	p Max p: 53	Hexadecimal floating-point	p≤21: Word p>21: Doubleword	The data is normalized in storage before and after arithmetic operations.
Float	Decimal	p Max p: 16	Hexadecimal floating-point	p≤6: Word p>6: Doubleword	The data is normalized in storage before and after arithmetic operations.
COMPLEX data					
Fixed	Binary	p,q Max p: 31	Fixed-point binary	Word	As for real fixed binary. The real and imaginary parts occupy adjacent fullwords, with the real part first.
Fixed	Decimal	p,q Max p: 15	Packed decimal	Byte	As for real fixed decimal. The real and imaginary parts occupy adjacent fields, with tne real part first.
Float	Binary	p Max p: 53	Hexadecimal floating-point	p≤21: Word p>21: Doubleword	As for real float binary. The real and imaginary parts occupy adjacent fullwords or doublewords, depending on the precision, with the real part first.
Float	Decimal	p Max p: 16	Hexadecimal floating-point	p≤6: Word p>6: Doubleword	As for real float decimal. The real and imaginary parts occupy adjacent fullwords or doublewords, depending on the precision, with the real part first.

Note: When p is even, the effective precision for all arithmetic operations except division is (p + 1,q), except when the SIZE condition is being checked. When this occurs, the first digit in the high-order byte must be checked to ensure that it is zero.

Figure 2. Arithmetic Data Representaton

Data type	Implementation		
	Representation	Length	Alignment
Bit	1 binary digit per bit	Maximum length: 32,767. If a VARYING attribute is declared, maximum length is declared length, regardless of the string value.	Byte (see note)
Character	1 character per byte		Byte

Note: The string occupies CEIL (n/8) bytes. If the string comes within the scope of an UNALIGNED attribute, the address of the first bit is provided by a byte address and bit offset in an SDV. (See 'String Dope Vector' in Appendix H.)

•Figure 3. String Data Representation

```
 0     7 8                     31
┌───────────────────────────────┐
│       Invocation Count        │
├───────┬───────────────────────┤
│       │   A(Statement label)  │
└───────┴───────────────────────┘
```
Figure 4. Statement-Label Data Representation

COMMUNICATION CONVENTIONS

The use of library modules in a PL/I program requires that:

1. Working storage be provided for the modules.

2. Techniques for passing information about arguments and program status be provided.

Working storage is obtained as <u>library workspace</u> (LWS). Appendix H gives the format of LWS, which is allocated by the library program management module IHESA.

Two modes of communication are available for passing information:

<u>Explicit:</u>	Uses parameter lists and registers. (See 'Linkage Conventions')
<u>Implicit:</u>	Uses <u>pseudo-registers</u> or a <u>Library communication area</u>.

Some library modules are interpretive (as opposed to declarative), and accordingly require that information regarding the characteristics of their arguments be supplied. Such information is made available to the library in the form of standardized control blocks. The form and content of the compiler-generated control blocks in general use throughout the implementation are described in Appendix H; one or more blocks is required according to the nature of the data passed:

Scalar arguments:
- Data element descriptor (DED)
- String dope vector (SDV)
- Symbol table (SYMTAB)

Array arguments:
- Array dope vector (ADV)
- String array dope vector (SADV)

Structures:
- Structure dope vector
- Dope vector descriptor (DVD)

Formats:
- Format element descriptor (FED)

Special-purpose control blocks, such as the file control block (FCB), are described in Chapters 3, 4, and 5, and in Appendixes I, J, and K.

<u>Pseudo-Register Vector (PRV)</u>

This is an area of task-oriented storage, addressed through register PR. The PRV contains a number of pseudo-registers which effectively operate as implicit arguments and give information about, for example, current program status. All references to specific pseudo-registers within the PRV are made by the addition of a fixed displacement to the PRV base address in register PR.

A pseudo-register is defined within a library module as a Q-type address constant which is fixed during the linkage editing process. All pseudo-register address constants within the PL/I implementation are two bytes long. The maximum size of a PRV is 4096 bytes. The pseudo-registers used by the PL/I Library are shown in Appendix C.

Library Workspace (LWS)

Various library modules require working storage:

1. For internal functions.
2. For linkage to other modules. (A register save area must be provided.)

Since the library is designed to function within a multitasking environment, such storage must be allocated on a task-oriented basis. The storage so allocated is termed library workspace (LWS).

Library modules which use LWS refer to it by means of the PRV. A group of pseudo-registers in the PRV is set during LWS allocation to contain the addresses of contiguous areas within LWS. (See Appendix H.) Each of these areas is at a different level.

The notion of level exists because of inter-module linkage between library modules:

1. A module which invokes no other modules is assigned level 0.
2. A module which invokes other modules is assigned a level number greater than the level number of any invoked module.
3. A module which transfers control to another module (i.e., does not expect a return) is assigned the level number of that module.

Invocation of the error-and-interrupt-handling subroutine is not considered sufficient to raise the level number of the invoking module, since the error subroutine uses a special level.

Library workspace is allocated as primary or secondary LWS.

Primary LWS is allocated during program initialization, before control is passed to the main procedure. The storage thus obtained is not freed until the PL/I program is finished.

Secondary LWS is allocated for special purposes during program execution and is freed when the situation for which it was created no longer exists. It is allocated:

1. When an on-unit is entered from a library module. This may lead to a recursion problem: library modules called may overwrite this LWS. To avoid this, the existing LWS is stacked, a new one obtained and all the LWS pseudo-registers updated.
2. When SNAP, system action or error messages are to be printed. The PRINT subroutine may overwrite the existing LWS: to avoid this, the same procedure is followed as for an on-unit.

The library program management module IHESA controls the allocation of LWS and the setting of the library pseudo-registers. (See Chapter 4.) The library macro IHELIB controls the length of LWS and of each area within it. The LWS format can be changed by changing IHELIB and reassembling IHESA.

Modules using specific areas in LWS address these areas by the following library macros:

IHEPRV: Used to address the LCA or when using an area as temporary workspace.

IHESDR: Used when a module requires a standard save area for a module it is calling.

Library Communication Area (LCA)

Within the area allocated for library workspace is an area in which various symbolic names are defined. These names are used for implicit communication between library modules (mainly the data conversion modules). This area is the library communication area (LCA); its format and the usage of the symbolic names are shown in Appendix H. The LCA address is stored in the pseudo-register IHEQLCA.

In the LCA there is a doubleword immediately before the first symbolic name. This contains (in the first four bytes) the address of the prior generation of LCA within a given task. This field is used to readdress the LCA which existed before an ON block was entered. IHEQLCA contains the address of the first symbolic name.

Object-Time Dump

A PL/I user may obtain a dump at any time by calling one of the following:

IHEDUMC:	Dump current task and then continue execution.
IHEDUMJ:	Dump all tasks and then continue execution.

IHEDUMP: Dump all tasks and terminate major task (i.e., terminate the job step).

IHEDUMT: Dump current task and then terminate it.

Identification of required information (such as save-area locations) in the dump is difficult because this information is not necessarily stored in locations arranged in a chronological sequence. To facilitate reading the dump, therefore, two subroutines, IHEZZC and IHEZZF, are provided. They extract certain information (chiefly about save areas and opened files) and print it as an index to the dump. Full details of this information are given in Appendix F.

If a DD card exists, the information will be printed on the PL1DUMP file (unless there is something wrong with the PL/I save-area chains, in which case the SYSABEND or SYSUDUMP file will be used). If the data set specified is other than the SYSOUT file, DISP=MOD should be used on the DD card. If there is no DD card and the operating system has the primary control program or MFT, only the normal indicative dump will be provided; in an MVT environment, if there is no DD card, there will be no dump at all.

Checkpoint/Restart

| In an operating system with PCP or MVT, a PL/I user may establish a checkpoint at any point within a job step by calling IHECKPT. He must include a DD statement with the ddname SYSCHK to define the data set on which the checkpoint information is to be saved.

The module IHECKP is called directly from compiled code. It obtains an ordinary VDA for use as a save area, rather than using library workspace, because the CHKPT macro instruction that is issued by IHECKP makes use of the first byte of the save area; the first byte of a save area in LWS is used for PL/I information. (Refer to Chapter 4 for a discussion of the VDA and LWS VDA.) Each time IHECKP is called, it creates, from a dummy held as part of the module, a DCB that refers to the data set defined in the SYSCHK DD statement; on return from the CHKPT routine, the DCB is freed. The address of the DCB is the only parameter passed to the CHKPT routine.

SORT/MERGE - PL/I Interface

A PL/I procedure may call the operating system SORT/MERGE program, using the library module IHESRT. The publications in which the operation of SORT/MERGE is described are: IBM System/360 Operating System: SORT/MERGE, Form C28-6543, and, SORT/MERGE Program Logic Manual, Form Y28-6597.

Four entry points, IHESRTA, IHESRTB, IHESRTC, IHESRTD are provided to enable use to be made of SORT/MERGE user exits E15 and E35 to call PL/I procedures, as required by the application.

SORT/MERGE control statements are supplied as arguments to the PL/I CALL statement. These arguments correspond in format to standard SORT/MERGE control statements, from which the parameter lists are generated.

These arguments also specify the PL/I entry points to be invoked by the user exits E15 and E35, and any return codes to be used for inter-program communication.

The normal library conventions for save-area chaining are not used for this module. Instead the module allocates a DSA (with code X'80' in the first byte). This is to ensure that if either user exit is used, the chain-back is through the DSAs only.

After the parameter list for SORT/MERGE is generated, the following actions are performed before linking to SORT/MERGE:

1. The registers in the external save area of the PL/I procedure are saved and replaced by special registers which are used in terminating the sort when:

   a. A PL/I exit procedure is terminated, due to an error, before the sort has terminated, or

   b. A GO TO from an exit procedure to a procedure at a level equal to, or higher than, the calling procedure, occurs.

   Otherwise the PL/I procedure would terminate allowing the operating system to regain control, either directly or indirectly, while the link to SORT/MERGE is still operative, with a resultant system interrupt. The registers stored in the special save area cause the calling procedure to enter IHESRT and complete the SORT/MERGE operation. Any user exit calls to the now non-existent PL/I exit procedures are deleted, before restoring the external save area and returning control from the PL/I procedure.

2. The PICA is set to system action for program interrupts.

3. Register 13 is set to a special save area with a chain back address of zero.

On normal completion of the sort, the PICA and external save area are reset to the conditions at entry to IHESRT and control is returned to the calling program.

If an exit is taken, the PL/I environment is reestablished and register 13 is reset to the DSA allocated for IHESRT. The exit procedure is then invoked and thus the DSA chain is correct.

Before returning to SORT/MERGE the PICA and register 13 are reset to their values on initial entry to the exit routine in IHESRT.

FILES AND DATA SETS

Within this publication, the term 'data set' refers to a collection of records that exist on an external device. A file is known as such only within a program; it is possible that, within a given program, several files will use the same data set concurrently (direct access only). Similarly, a data set may be used by several programs, either concurrently or successively.

The relationship between a file and a data set is established when the file is opened. The data set to be associated with a file is identified by the TITLE option. If this option is omitted or an implicit open occurs, a default identifier is formed from the first eight characters of the file name. The data set identifier is not the data set name, but the ddname (i.e., the name of the DD statement). Error messages which are related to file operations use the full file name (1 through 31 characters).

The attributes of a file in some instances restrict the attributes of its associated data set, but in those instances where device independence is possible, the full capabilities of the job control language DD statement are available. Unit assignment, space allocation, record format and length, and various data management options (such as write-verify) are established on a dynamic basis.

FILE ADDRESSING TECHNIQUE

In order to accommodate reentrant usage of a PL/I module, which may imply that the module exists in read-only storage, the following technique is employed to communicate file arguments. All calls from compiled modules to the library involving file arguments address a read-only control block, the DCLCB. The library, using a field within this control block, is able to address a cell within the pseudo-register vector generated for the task. This cell, the file register, in turn addresses a dynamically allocated control block, the file control block (FCB). (See Figure 5.)

Declare Control Block (DCLCB)

This control block, generated during compilation, contains information derived from a file declaration (either explicit or contextual). In addition, it contains the offset within the PRV of the file register, a fullword pseudo-register employed within the file addressing scheme. This pseudo-register contains the address of a dynamic storage area containing a file control block. The DCLCB is read-only, and thus permits compiled programs to exist within a reentrant environment (which may imply that the program is loaded into supervisor protected storage). The maximum length of a DCLCB is 56 bytes.

File attributes specified within the DCLCB may be supplemented, but not overridden, by attributes specified in the OPEN statement which opens the file. An excep-

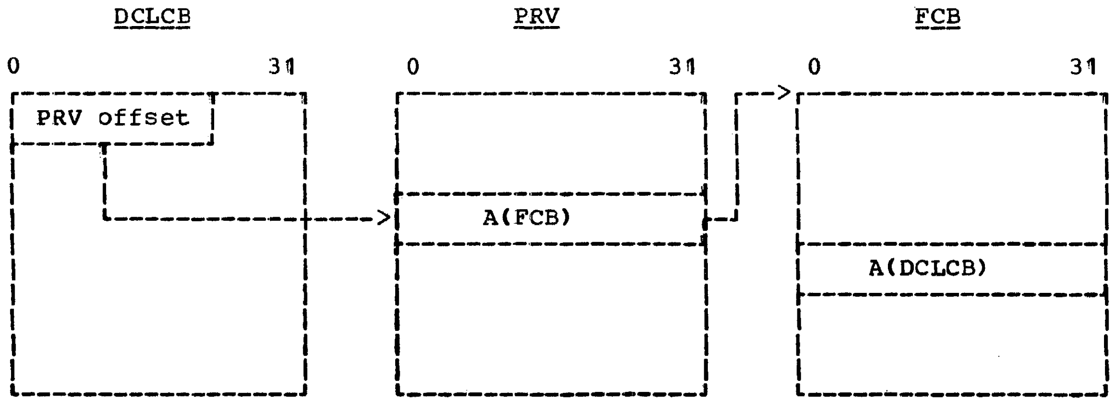


Figure 5. File Addressing Scheme

tion to this rule is the LINESIZE option, which overrules record length information declared in the ENVIRONMENT attribute.

The format of the DCLCB is described fully in Appendix I.

File Control Block (FCB)

This control block is generated during program execution when a file is opened. Dynamic allocation of the FCB storage is required in order to accommodate reentrant usage of a given module, for the FCB is not read-only. The FCB contains fields for both the PL/I Library and for operating system data management. The initial portion of an FCB is PL/I-oriented, while the second portion is the DCB required by data management for all data set operations. The PL/I portion, called the DCB-appendage, is described in Appendix I; details of the various DCB constructions are available in the following IBM publications:

IBM System/360 Operating System: System Control Blocks

IBM System/360 Operating System: Supervisor and Data Management Services

IBM System/360 Operating System: Supervisor and Data Management Macro Instructions

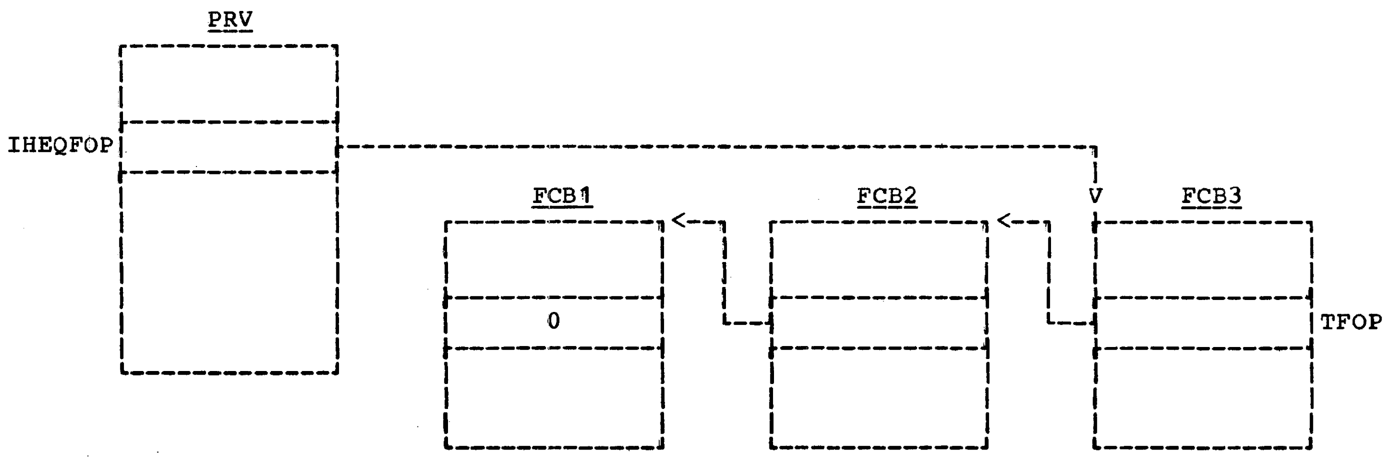
IBM System/360 Operating System: System Programmer's Guide

An FCB is generated for each file opened within a program; an FCB cannot exist for an unopened file. FCBs are generated in **task-oriented storage (in the same subpool as the PRV for the task: subpool 1).**

Accordingly, if a file is implicitly closed because of the termination of the task that opened it, its FCB is freed and the file register is set to zero. The contents of a given file register in a non-opening upward task are zero. Subsequent reference to the file may cause the file to be reopened. (A non-opening upward task for a given file is a task that does not open the file, and which is not a subtask of a task that has opened the file.)

When a file is opened, its generated FCB is placed in a chain which links together (through the TFOP field in the FCB) all files opened in a given task. When files are closed, they are removed from the chain. This chain, which is anchored in the PRV cell IHEQFOP, exists in order to perform special PL/I closing processes at task termination (whether normal or abnormal). When a task terminates, the object-program housekeeping routines determine which files are currently opened by this task. This is performed by the relevant housekeeping module calling IHEOCLD (close), which scans the chain and calls IHECLTB to close all files opened in the current task. If the cell IHEQFOP is zero, then no files are, at present, opened by the task. When a subtask is attached, this cell is initialized to zero in the newly generated PRV. The IHEQFOP chain is shown in Figure 6.

Since an FCB is generated in dynamic storage, its address cannot be determined either at compile time or link-edit time; it is this characteristic of the FCB which requires the file addressing scheme out-



Note: **The FCBs are opened in the order 1, 2, 3, etc.**

Figure 6. Format of the IHEQFOP Chain

lined above. If a given procedure is being executed by two or more jobs (multi-jobbing), an FCB (with its associated PRV) exists for each job; the procedure does not, however, necessarily operate on different data sets. Similarly, if a file is opened in two parallel subtasks, an FCB exists for each task.

Program Execution

When program execution is initiated, the PRV (including all file registers) is initialized to zero. When a file is opened (prepared for I/O operations), its associated file register is set to address an FCB; similarly, when a file is closed explicitly, its file register is again set to zero.

Since a copy of the PRV of the attaching task (calling procedure) is provided to the attached task (called procedure), the state of a file is communicated downward through major to minor tasks. If the file is not open, the file register remains zero. If a file has gone through the opening process but has failed to be opened (UNDEFINEDFILE condition), the high-order byte (bits 0 to 7) of the file register will contain an error code that indicates the cause of failure. The codes consist of two hexadecimal digits; they are shown in Figure 7.

If the file register is non-zero, the file is open and its FCB is also available to all the subtasks created while the file was in the open state. This technique of communicating the state of a file makes it possible to access a file in two parallel subtasks.

Two advantages of the use of the DCLCB in the file addressing scheme are:

1. Because the DCLCB, in conjunction with an implicit opening statement, provides all the information necessary to open a file, a file can be opened by I/O statements other than the OPEN statement.

2. Because the DCLCB is part of the static storage of a load module, its address is constant throughout program execution. This address can be used therefore as the file identification in ON conditions that relate to files. ON conditions may be enabled for a file before it is opened, since the DCLCB address is always available.

Error code	Meaning
81	Conflict between DECLARE and OPEN attributes
82	File access method not supported
83	No block size
84	No DD card
85	TRANSMIT condition while initializing data set (only applicable to DIRECT OUTPUT REGIONAL files)
86	Conflict between PL/I attributes and environment options
87	Conflict between environment options and DD parameters
88	Key length not specified
89	Incorrect block size or logical record size specified
8A	Line size greater than implementation-defined maximum

Figure 7. Error Codes Indicating Causes of Failure in Open Process

OPEN/CLOSE FUNCTIONS

The opening of a file occurs either explicitly by the use of an OPEN statement, or implicitly because of other I/O operation statements.

Opening a file involves the creation, within dynamic storage (subpool 1 of the opening task), of an FCB, the setting of a file register to address the FCB, and the invocation of the data management OPEN executor. The closing of a file involves invocation of the data management CLOSE executor, freeing FCB storage, and clearing the associated file register.

EXPLICIT OPENING

In order to conserve storage, the module structure of the OPEN and CLOSE processors involves a 'bootstrap' routine, IHEOCL, which links to the modules IHEOPN and IHECLT. In a multitasking environment IHEOCT links to IHEOPN and IHECTT. The bootstrap module passes to the loaded modules the address of a list of all necessary address constants and pseudo-register offsets, since these cannot be set in a module not link-edited with the executing program. The list is found in the library module IHESA (non-multitasking) or IHETSA (multitasking).

All errors are communicated back to IHEOCL/IHEOCT by means of the file registers; IHEOCL/IHEOCT then invokes the error handling subroutine. The error conditions are signaled in the high-order byte of the file register; IHEOCL/IHEOCT, upon detecting an error condition, sets bit 0 of this register to indicate an unopenable file. The error codes are shown in Figure 7.

Open Control Block (OCB)

One of the parameters which may be passed to IHEOPN is the open control block (OCB), which is generated by the compiler. This four-byte control block indicates the attributes specified in the OPEN statement. During the opening process, this information is merged with that in the DCLCB in order to construct the proper FCB and check for attribute conflicts. (See Appendix I for details of the OCB.)

The Open Process

The flow through the OPEN modules is illustrated in Figure 8.

The open process is performed by the modules IHEOPN, IHEOPO, IHEOPP, IHEOPQ and IHEOPZ which reside within the LINKLIB data set. These modules are dynamically loaded in order to conserve object-program storage. They initially receive control from a bootstrap module, IHEOCL (non-multitasking) or IHEOCT (multitasking); each module, after performing its functions for all files being opened, passes control to the next by the XCTL macro. IHEOPQ then returns to the bootstrap module.

Open Process, Phase I: IHEOPN: This performs file attribute checking and defaulting functions. If a file being opened is REGIONAL, and is opened for DIRECT OUTPUT (creation), the module IHEOPZ is invoked by IHEOPN to initialize (format) the initial space allocation of the associated data set. Such initialization is required in order to allow subsequent direct insertion of records into the data set. If, in phase I, all files specified in the OPEN statement have detected errors, a return to the bootstrap IHEOCL is made immediately. Otherwise phases II, III and IV are invoked and a return is made to IHEOCL from IHEOPQ.

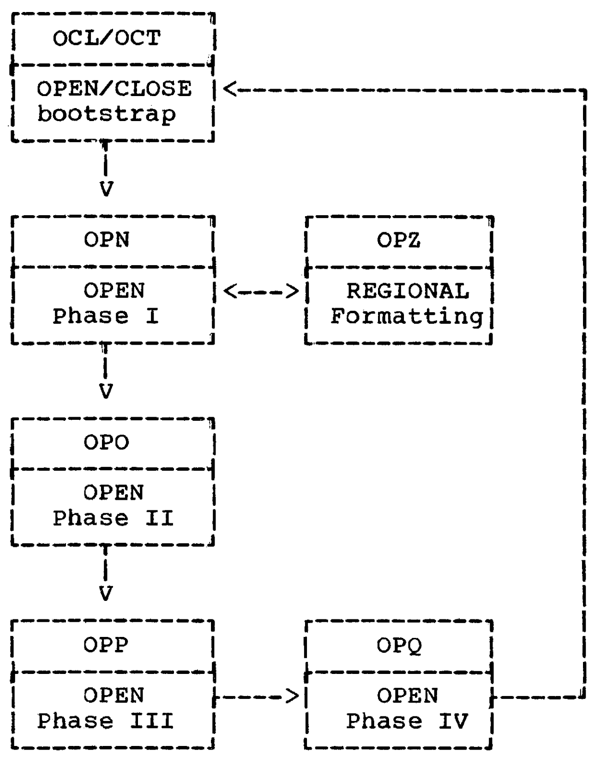


Figure 8. Flow through the OPEN Modules

Initialization for REGIONAL data sets of F format records involves writing dummy records (and keys, except for REGIONAL (1)) throughout the data set. On the other hand, initialization for U or V format records (REGIONAL (3) only) requires merely that the capacity record (R0) be written in each track to signal a free track, the track being automatically cleared as well.

Open Process, Phase II: IHEOPO: This obtains storage for an FCB for each file being opened, and sets fields in both the DCB and the DCB-appendage according to the declared attributes.

Open Process, Phase III: IHEOPP: This executes the OPEN macro, and accepts DCB-exits.

Open Process, Phase IV: IHEOPQ: This dynamically loads record-oriented I/O modules (setting their addresses in the FCB), and enters the files being opened into the IHEQFOP chain of files opened in the current task.

The Close Process

This process consists of: removing files from the IHEQFOP chain; freeing dynamically acquired storage (file control blocks, buffers, exclusive control blocks, and I/O control blocks); and deleting any appropriate dynamically-loaded record-oriented I/O modules. In the following description the non-multitasking module is followed with its multitasking alternative in parentheses.

Module IHEOCL (IHEOCT) starts the close process; for an explicit close it links to IHECLTA (IHECTTA); for an implicit close to IHECLTB (IHECTTB). If the last operation on a BUFFERED SEQUENTIAL INDEXED OUTPUT embedded-key file, before it is closed explicitly, is LOCATE, module IHEOCL (IHEOCT) replaces the embedded key with the KEYFROM option, before passing control to IHECLT (IHECTT). For further information refer to Indexed Data Sets on page 35.

Module IHEOCL (IHEOCT) calls IHEITC to finish formatting the current extent when closing a REGIONAL SEQUENTIAL OUTPUT file. If IHEITC finds a key sequence error due to a previous LOCATE statement on a REGIONAL file with U- or V-format records the key sequence is ignored and a message is displayed on the console.

The normal return from a KEY on-unit is to the statement following that in which the condition is raised. Consequently, if the KEY condition is raised during the execution of an explicit CLOSE statement, the file will not be closed unless the on-unit also includes a CLOSE statement.

In addition, if a file is closed implicitly (on termination of a task), IHEOCL or IHEOCT scans the IHEQFOP chain to find the file. In a multitasking environment, if a task is terminated normally, IHEOCT unlocks all records locked in the task and frees the corresponding exclusive blocks; if a task is terminated abnormally, it merely removes the exclusive blocks from their chains. For an implicit close, all events associated with event variables in the IHEQEVT chain are purged, and the associated IOCBs, if any, are freed.

Modules IHECLT and IHECTT reside within the LINKLIB data set and are loaded dynamically in the same manner as the OPEN modules. They perform additional special functions as follows:

Stream-oriented I/O:

If OUTPUT with U-format records, the last record is written.

Record-oriented I/O:

All incomplete event variables associated with the file are set complete, abnormal, and inactive, and the I/O operations are purged.

In a multitasking environment:

1. The event variables in the TEVT chain are set complete, abnormal, and inactive.

2. For a REGIONAL EXCLUSIVE file, or an INDEXED EXCLUSIVE file with unblocked records, locked records are unlocked and all exclusive blocks in the TXLV chain are freed.

3. For an INDEXED EXCLUSIVE file with blocked records, the file is unlocked.

IMPLICIT OPENING

If a file is not open and an I/O operation is initiated, then one of the compiler-interface modules (IHEIOA, IHEIOB (or IHEIBT), or IHEION (or IHEINT)) calls IHEOCL (or IHEOCT), at implicit-open entry point IHEOCLC (or IHEOCTC), passing any implied parameters, and the open process begins.

If the OPEN modules return control to IHEOCL (or IHEOCT) and the file is still unopened, the UNDEFINEDFILE condition is raised.

STREAM-ORIENTED I/O

Although I/O devices available within IBM System/360 Operating System are usually designed to transmit data in records of various lengths (blocks), the stream-oriented facilities allow a program to ignore record boundaries. The GET and PUT statements transmit data between storage and one or more records which exist within a buffer, the location within the buffer being updated as each data field is accessed. When a record becomes filled (if output) or empty (if input), another record is obtained. Support for record access is provided by the data management access method QSAM (queued sequential access method). Normally, the GET and PUT data management macros are used in the locate mode, to conserve space and time; paper tape input, however, must use the MOVE mode. The flow through the stream-oriented I/O modules is shown in Figure 10.

CURRENT FILE

The current file is that one which is being operated upon by an I/O statement; it is established when an operation begins, and removed when the operation is completed. The current file is addressed through the pseudo-register IHEQCFL, which addresses the DCLCB for the file. This pseudo-register is available for inspection upon entry to ON blocks, and during transmission. Its format is shown in Figure 9.

0 7	8 31
0	A(DCLCB)
	A(Abnormal return)

Figure 9. Format of the Current File Pseudo-Register

Within a stream-oriented data specification there may exist expressions which involve function references. In turn, the function procedure may itself perform I/O operations or may refer to ON blocks that perform I/O operations. When this situation occurs, it is necessary to stack the current file pseudo-register. The presence of the COPY option in a GET statement and the raising of the TRANSMIT condition for an <u>item</u> in the data stream are flagged in the fifth byte of IHEQCFL:

TRANSMIT to be raised on <u>item</u>:	Bit 5 = 1
COPY option in statement:	Bit 6 = 1
Current file in PRV:	Bit 7 = 0
Current file stacked in DSA:	Bit 7 = 1

Stacking of the current file is effected by the I/O initialization modules; upon entering such a module (e.g., IHEIOA and IHEIOB), the contents of the pseudo-register IHEQCFL are stored in the DSA (dynamic storage area) of the invoking procedure, as addressed by register DR. The stacking cell is termed the current file pseudo-register update. (See Chapter 4.) Upon termination of an I/O operation, either normally, or by means of a GO TO statement out of an ON block, this cell is copied back into the pseudo-register IHEQCFL.

GET and PUT statements with the STRING option employ the current file pseudo-register, but no abnormal return entry exists. Instead, the latter four bytes address a simulated FCB.

STANDARD FILES

The standard files, SYSIN and SYSPRINT, have default titles equivalent to their file names. The compilation of GET and PUT statements without explicit FILE options causes compile-time syntax substitution of the file names SYSIN and SYSPRINT respectively. These files have the same compiled linkage to the library as other files. Within the library, SYSIN is not used; the file SYSPRINT, however, is used in that error messages and listing of data fields for the COPY and CHECK options require the presence of this file.

SYSPRINT may be implicitly opened either by:

1. the first PUT executed in the compiled procedure, or

2. a call from within the library for the COPY option or an error message.

If the library attempts to open this file, and it cannot be opened (missing DD card, etc.), this situation is flagged and all error messages will appear on the system console. In addition, any COPY options, or system action for the CHECK condition, will be ignored. The UNDEFINEDFILE condition is suppressed in the above cases.

If a compiled procedure attempts to open SYSPRINT, and it cannot be opened, the normal UNDEFINEDFILE condition is raised.

Because the library and the source program both use the SYSPRINT file, it is necessary that they both refer to the same DCLCB. This is achieved by the use of CSECT facilities within the linkage editor; both the compiled DCLCB and the library-supplied DCLCB for SYSPRINT (within the module IHEPRT) are supplied with the same name, so that only one of them will be placed within the linked program. The name of both CSECTs is IHESPRT; the name of the associated file register is IHEQSPR.

SYSPRINT IN MULTITASKING

In a multitasking environment, to ensure that there is no conflict between operations in different tasks that refer to the same non-exclusive file, it is necessary for the programmer to synchronize these operations (by using an EVENT variable, the COMPLETION pseudo-variable, and the WAIT statement). Since the library uses the file SYSPRINT, it is not possible for the programmer to synchronize all operations on this file. Therefore the library

<u>Note:</u> An asterisk indicates that the module can be entered directly from compiled code

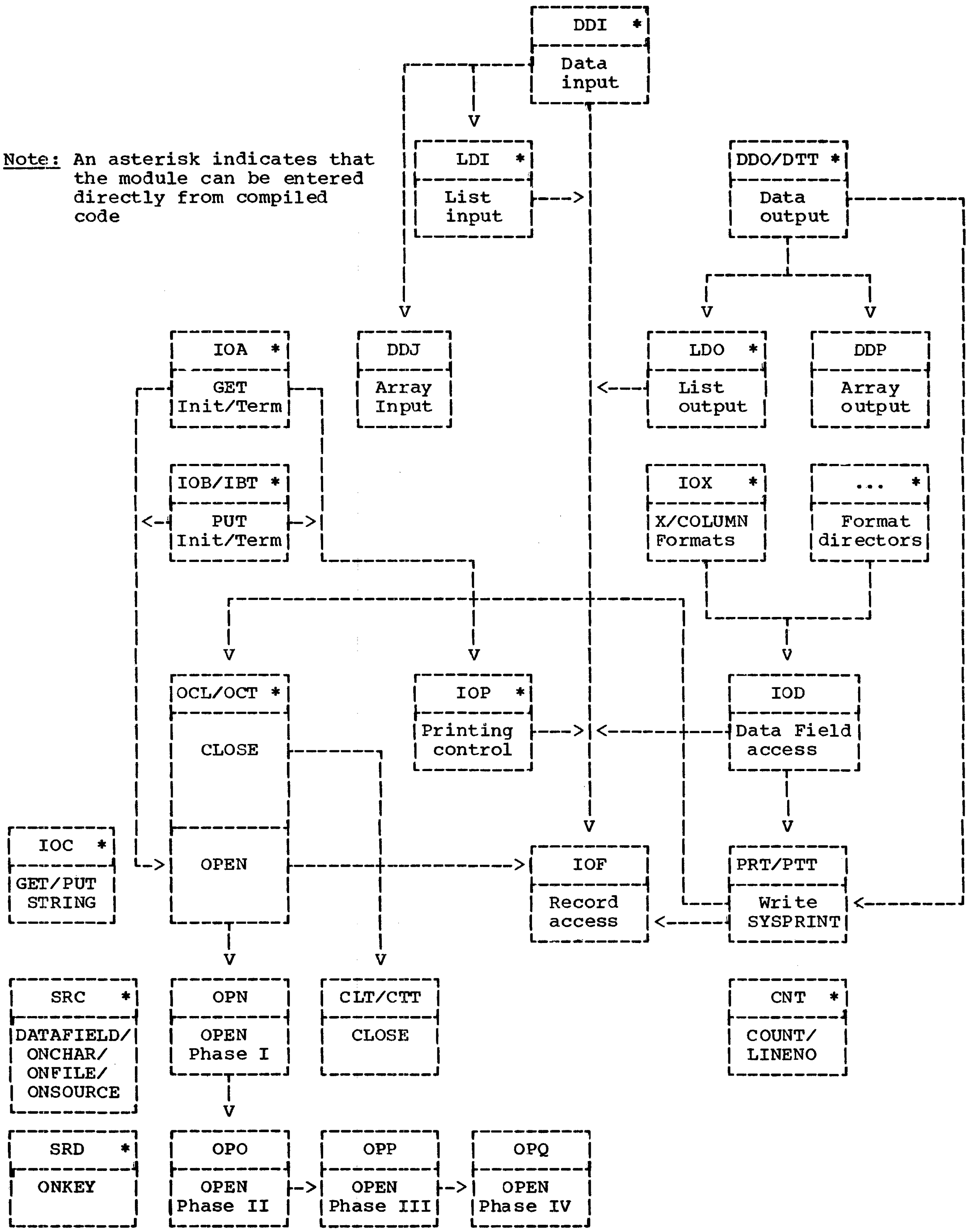
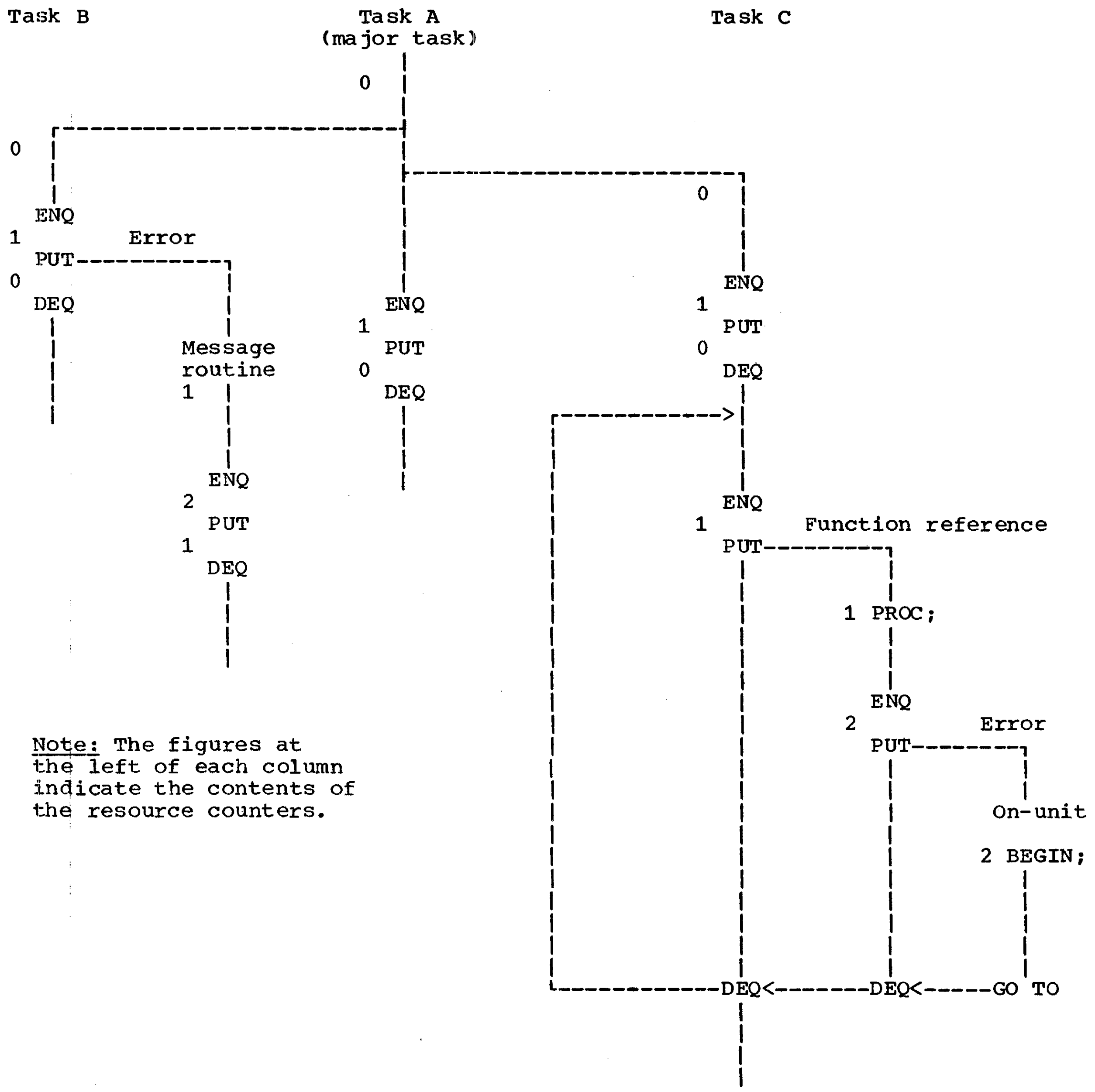


Figure 10. Modular Linkage through Stream-Oriented I/O



Note: The figures at the left of each column indicate the contents of the resource counters.

Figure 11. Allocation of SYSPRINT Resources in Multitasking

module that implements PUT statements for SYSPRINT (IHEIOB), and other modules that use this file, issue an ENQ macro instruction before executing each PUT statement on SYSPRINT, and a DEQ macro instruction on completion of the operation. All SYSPRINT operations cannot be enqueued on the same resource, since this could result in an interlock situation (two or more operations, each waiting for the completion of the others). For example, this would be the case if a PUT statement involved a function reference that required another PUT operation; if both were enqueued on the same resource, the second operation could not commence until the completion of the first, which itself could not proceed until the function had returned an answer.

The library resolves the difficulty by employing a resource counter (the first byte of the current-file field in the DSA: see Appendix J). Before each SYSPRINT operation is executed, the operation is enqueued on the resource number in the counter, and the counter is then incremented by one; on completion of the operation, the counter is decremented by one before the operation is dequeued. When a new DSA is obtained (on entry to a new block: see Chapter 4), the resource count is copied from the DSA of the block from which the new block was entered.

In the example (Figure 11), when the major task (task A) is initialized, the resource count in its DSA is set to zero. Task A then attaches tasks B and C, and in each case the resource count (0) is copied into the new DSA. Tasks A, B, and C then request PUT operations, all of which are enqueued on resource 0; in each case the resource count is then incremented by 1. These operations are therefore completed in the order in which they were requested.

During execution of the PUT statement in task B, an error condition occurs that involves a library call to print a message (e.g., UNDERFLOW). The library PUT statement is enqueued on resource 1, since the resource counter is incremented after the task PUT statement is enqueued, but before the statement is executed. The library PUT operation is therefore not dependent on the completion of the PUT statement that raised the error condition.

If a GO TO statement is executed that passes control to a statement preceding a series of enqueued operations, the program management routine IHETSAG releases the DSAs of the blocks thus freed and dequeues the I/O operations they contain. This is illustrated in task C (Figure 11), where control is passed to an on-unit as a result of an error in a PUT statement in a function reference made during the execution of the second PUT statement in the task. The PUT statement is enqueued on resource 0, and the resource count is then incremented. When the function is called, the resource count (1) is copied into its DSA; consequently, the next PUT statement is enqueued on resource 1, and the counter is again incremented. The count 2 is copied into the on-unit DSA when control passes to the on-unit. On execution of the GO TO statement, which passes control back to a statement preceding the original PUT statement, IHETSAG frees the function and on-unit DSAs, dequeues all the PUT operations, and resets the resource counter in the DSA for task C to its value on entry to the task (0).

No special provision is made for handling SYSPRINT resources on termination of a task, since this file cannot be used by the library end-of-task exit routine.

The qname and rname used in the ENQ and DEQ macro instructions are:

```
    qname (two words):
        Bytes 1-4: A(SYSPRINT FCB)
        Bytes 5-8: A(SYSPRINT FCB)

    rname (1 byte):
        Resource count in DSA
```

GET/PUT OBJECT PROGRAM STRUCTURE

The code compiled for stream-oriented I/O GET and PUT statements has the general structure illustrated in Figure 12. There are three 'call sets' compiled for these statements:

1. Initialization:

   This call invokes one of the I/O initiator modules, passing:

   a. The address of the file DCLCB.

   b. The address of the termination call. (This is the abnormal return which is set within the current file pseudo-register IHEQCFL.)

   c. The address of the LINE or SKIP value.

   The initialization process includes stacking the current file, checking the specified file (and opening it if not already open), and performing any necessary option operations.

2. Data specification:

   This is a series of calls to perform list-, data-, or edit-directed stream-oriented I/O operations. This series is omitted only for GET/PUT statements which have no data specification. Details of the implementation of the three forms of data specification appear in 'Data Specifications', below.

3. Termination:

   This call invokes the terminal subroutine of the module which performed the initialization. At this point the current file is unstacked and (for PUT calls) V format output records have their record-length field updated.

DATA SPECIFICATIONS

There are three forms of data specification:

Data-directed

List-directed

Edit-directed

Compilation of any data specification yields a series of one or more calls to the library for transmission of data between program storage and a record buffer. For list- and data-directed I/O, the data items transmitted are passed by means of the standard linkage described above. (See 'Linkage Conventions' in Chapter 2.) The PL/I standard (using registers) is employed wherever possible; where it is not, the operating system standard (using a parameter list) is employed. For edit-directed I/O, the 'executable format scheme' described below is required.

The ON CHECK facilities for data items being input are supported by compiled code between data-list item specifications, in the instances of list- and edit-directed I/O; data-directed I/O determines the existence of this condition from the symbol table entry for a given data item.

EXECUTABLE FORMAT SCHEME

The executable format scheme exists to support two requirements for edit-directed data items:

1. The matching at object time of data-list items with format-list items.

2. The evaluation of expressions during an I/O operation.

The scheme exists in compiled code for use by the library format directors and conversion package. (See 'I/O Editing and Data Conversion' in Chapter 8.)

The scheme is required because edit-directed data specifications contain format lists composed of format items that may have expressions for replication factors and format subfields. These expressions may have to be evaluated with values read in during a GET operation. Finally, the use of dynamic replication factors and the possible existence of array data-list items of variable bounds prevent any predeterminable matching of data-list items and format-list items.

Basically, the scheme calls for the existence of two location counters, one for a compiled series of data-list item requests, the other for a compiled series of format-list item specifications. These two series are compiled as the secondary calling set for a GET or a PUT operation.

To support the dynamic matching of a format-list item with any data-list item, a group of format directors exists within the library; one of these directors receives the call from the secondary compiled series of format item specifications. A director will determine which conversions are required to satisfy the transmission of a data item according to its internal representation (described by its DED) and its specified external representation (described by a FED).

The structure of edit-directed compiled code is illustrated in Figure 13. The first column, 'Primary code', consists of calls to units in the second column, 'Secondary code'; i.e., data-list items are requesting a match with a format-list item. The third column shows the flow within the library as set up by format directors.

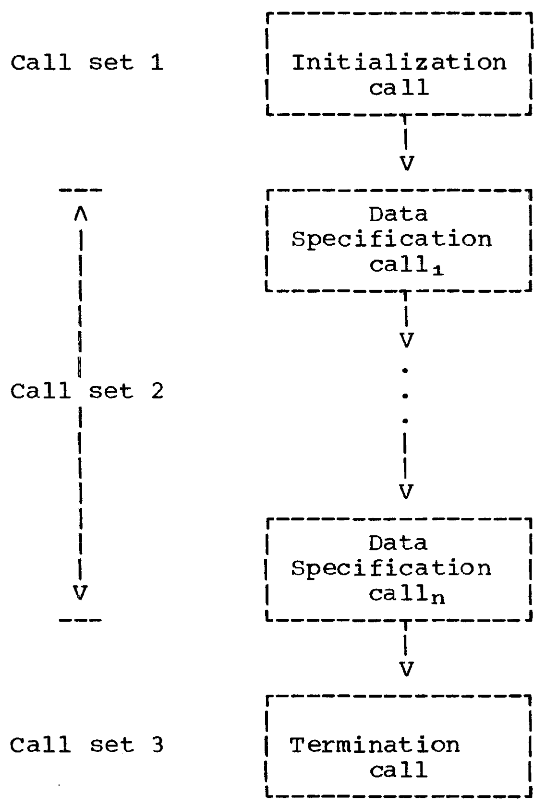


Figure 12. Object Program Structure of GET/PUT

The scheme works as follows:

1. The address of the start of the format-list code (executable format) is obtained.

2. Transmission of the first data item is requested; its storage address and DED address are loaded into registers RA and RB.

3. Control is transferred to the executable format; at the same time the

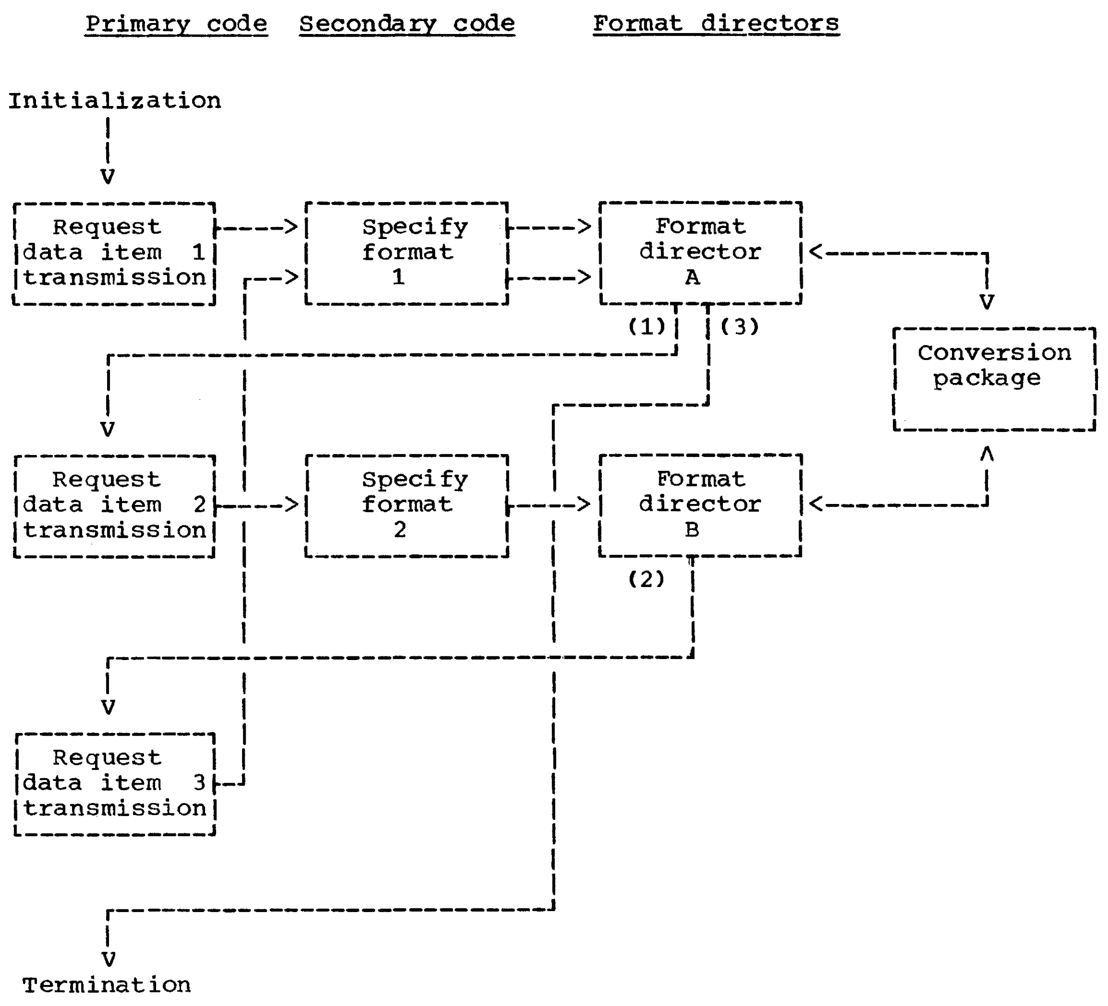


Figure 13. Executable Format Scheme

location counter of the data-list code is updated.

4. The executable format loads, into register RC, the address of an FED.

5. A call is made to a format director and at the same time the location counter of the format-list code is updated.

6. The format director causes the conversion package to convert the data according to DED and FED information, storing the converted data in the specified storage address, if input, or placing it in a buffer, if output.

7. Return is then made to the data-list code, by means of the data-list location counter, LR.

8. The above steps, 2 through 7, are repeated until the end of the data-list code is reached.

Within both primary and secondary code, looping and invocation of function procedures may occur. Within secondary code, the appearance of control format items (PAGE, SKIP, LINE, COLUMN, X) will cause the location counter for primary code, register LR, to be temporarily altered, so that control is returned from the library, not to the primary code, but to the secondary code. This allows the data-list item which activated the control format item to be matched with a data format item.

OPTIONS

COPY: This option causes each data field accessed during a GET operation to be listed on the standard output file, SYSPRINT. This is performed by calling the module IHEPRT. Each data field occupies the initial portion of a line.

If there is no DD card for SYSPRINT, the COPY is ignored by IHEPRT.

STRING: This option causes a character string to be used instead of a record from a file. This situation is made transparent to the normal operation of the I/O modules since the initialization module for GET/PUT STRING (IHEIOC) constructs a temporary FCB for the string. Information regarding the address and length of the string is set in the FCB fields TCBA, TREM and TMAX. A temporary file register is created in the second word of the pseudo-register IHEQCFL. (A dummy DCLCB is placed in front of the generated FCB and consists of two bytes which indicate the offset of the dummy file register.)

PAGE, SKIP, LINE (print files): These options cause the current record (which is equivalent to a 'line') to be put out, and a new record area to be obtained. SKIP can also be used with input to cause the rest of a record in the input stream to be ignored. Record handling for these functions is performed by the module IHEIOP. All printing options (and format items) are supported by use of the ASA control characters:

1	Page eject
+	Suppress space before printing
b	Single space before printing
0	Double space before printing
-	Triple space before printing

Should spacing greater than triple be required for a LINE or SKIP request, a series of blank triple space records is generated, followed by a single or double space record, if necessary.

SKIP (non-print files):

1. Input files: The SKIP(n) option causes the rest of the current line (record) to be ignored in the input stream, and a further (n - 1) lines to be ignored.

2. Output files: The SKIP(n) option causes the remainder of the current line (record) to be ignored and (n - 1) blank lines to be inserted into the output stream. Note that, for format F records, each line is padded with blanks; for format V and U records, only the necessary control bytes and record lengths are supplied.

Organization	Access	Mode	Buffering	Record Format	Access Method	Notes on Use of Access Method
CONSECUTIVE	SEQUENTIAL	INPUT OUTPUT UPDATE	BUFFERED	ALL	QSAM	Locate-mode (except paper tape)
			UNBUFFERED	F, U, V	BSAM	-
INDEXED	SEQUENTIAL	INPUT UPDATE	BUFFERED or UNBUFFERED	F, FB[1]	QISAM	Scan-mode; ESETL/SETL
		OUTPUT				Load-mode
	DIRECT	INPUT UPDATE	-	F, FB	BISAM	-
REGIONAL(1) REGIONAL(2) REGIONAL(3)	SEQUENTIAL	INPUT UPDATE	BUFFERED or UNBUFFERED	F (REGIONAL(1), REGIONAL(2))	QSAM/ BSAM[3]	-
		OUTPUT			BSAM	BSAM Load-mode
	DIRECT	INPUT OUTPUT UPDATE	-	F, U, V (REGIONAL(3))	BDAM	REGIONAL(1)[2] Relative record without keys REGIONAL(2)[2] Relative record with keys REGIONAL(3)[2] Relative track with keys

Note 1: FB is not allowed for UNBUFFERED files
Note 2: OUTPUT causes data set to be formatted using BSAM (BDAM load-mode) at open time
Note 3: QSAM is used for REGIONAL(1) BUFFERED but not KEYED

Figure 14. Data Management Access Methods for Record-Oriented I/O

RECORD-ORIENTED I/O

OBJECT PROGRAM STRUCTURE

In record-oriented I/O, the data entities accessible to the source program are data management logical records (unlike stream-oriented I/O, where the data entities are data fields).

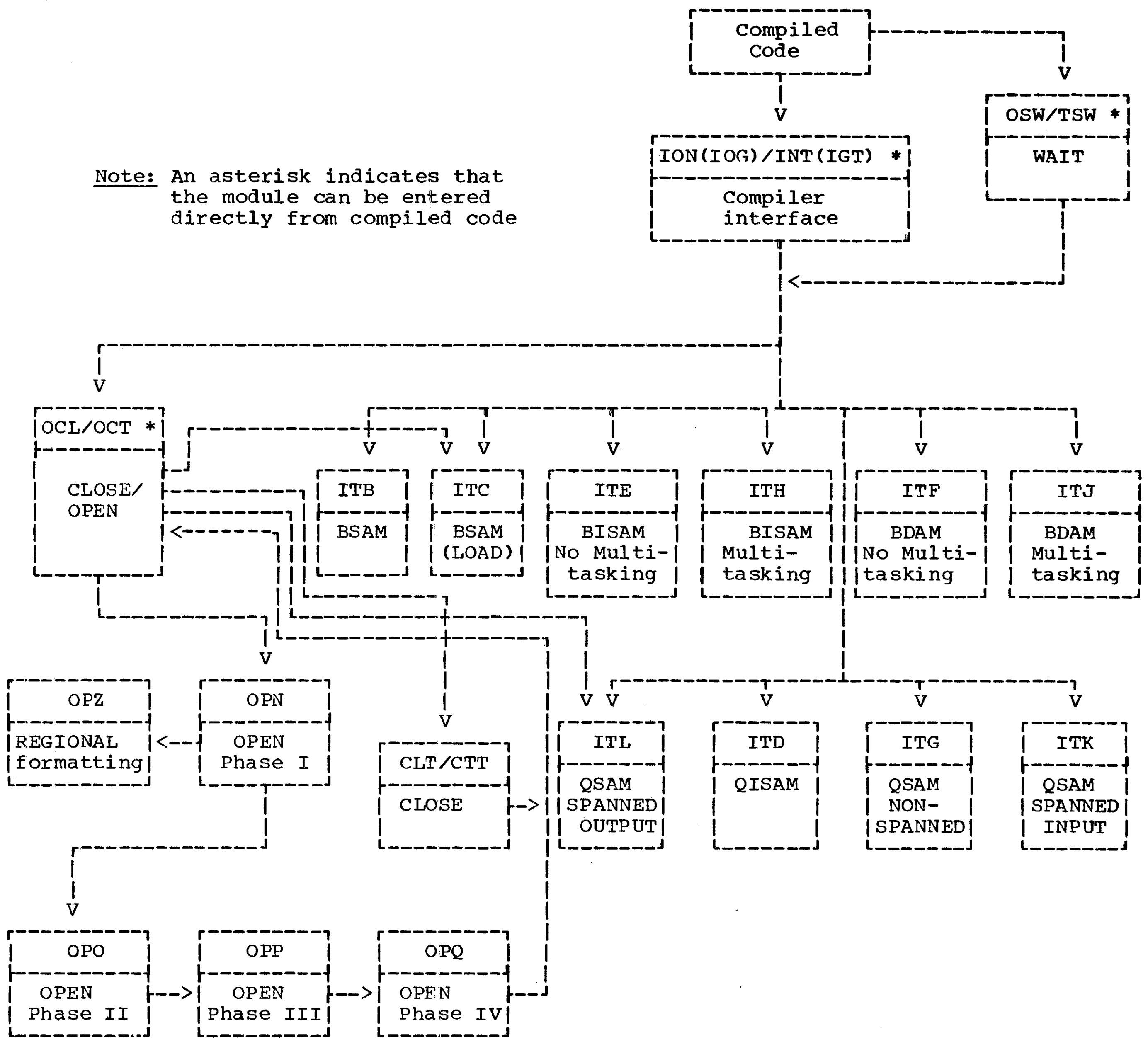
A wider range of record access is therefore available with record-oriented I/O: records may be keyed or not, may be directly or sequentially accessed, and may be manipulated within the data set by insertion, replacement, or deletion. The specific facilities available vary according to the data management access method employed to support a given data set.

The data management facilities employed are indicated in Figure 14, according to the organization of the data set. Note that not only the declared organization but also the mode of access and the format of records determine the chosen access method. Details of the manner in which the access methods are employed are provided in 'Access Method Interfaces'.

General Logic and Flow

The overall flow of record-oriented I/O modules is illustrated in Figure 15. Modules IHEION(IHEIOG) (non-multitasking) or IHEINT(IHEIGT) (multitasking) are general interface modules, one of which is invoked by a compiled call for any record-oriented I/O statement, in either a non-multitasking or multitasking environment. This module interprets the requested I/O operation, verifies its applicability to the specified file (and, possibly, implicitly opens it), and then invokes an access method interface module (characterized by the module names IHEIT*) to have the operation performed.

Modules IHEION and IHEINT supersede modules IHEIOG and IHEIGT at Release 17. The latter are retained in case a previously compiled load module is link-edited with the new library. The new modules perform


Note: An asterisk indicates that the module can be entered directly from compiled code

• Figure 15. Linkage of Access Modules in Record-Oriented I/O

the same function as the old except that they transfer control to the transmitters rather than link to them. The transmitters return direct to compiled code. This avoids saving and restoring registers between the interface module and the transmitter.

The verification of a statement is performed by IHEION (IHEINT in multitasking) by ANDing together a mask at offset -8 from the FCB and the second word of the Request Control Block. If the result is zero then the statement is invalid. The mask in the FCB is set up by IHEOPQ to indicate which statements are valid, and the RCB contains the statement type as a single bit in its second word.

On receiving control, the interface module first performs any necessary key analysis and record-variable length checking, and establishes any control blocks required. It then invokes data management for the transmission of a record. After transmission, or (if the EVENT option is employed) after initiation of transmission, control returns to the general interface module IHEION (or IHEINT), and thence to the compiled program. Errors may be detected within IHEION (or IHEINT) before an interface module is invoked, or within an interface module either before or after data management has been invoked. The relevant ON condition is raised when detected.

As indicated by the overall flow diagram, record-oriented I/O is implemented in such a fashion that the addition of further access method interface modules requires minimal changes (if any) within other parts of the implementation. The general interface module IHEION or IHEINT provides each access method interface module with a standard parameter set:

RA: A(Compiled parameter list)

Parameter list:

A(DCLCB)

A(Record dope vector/IGNORE/SDV)

A(Event variable)/0/A(Error return)

A(KEY|KEYFROM|KEYTO SDV)/0

A(Request control block)

The record dope vector and the request control block are described below under 'Record-Oriented I/O Control Blocks'.

The interface modules are also invoked to handle WAIT statements associated with I/O events. The WAIT module, having determined that an event variable (see Appendix I) is associated with a record-oriented I/O operation, invokes the relevant I/O transmitter (IHEIT*), passing the following parameters:

RA: A(Compiled parameter list)

Parameter list:

A(DCLCB)

A(IOCB being waited for)

A(Event variable)

(Reserved)

A(Request control block)

The transmitter then completes the previously initialized record transmission, and performs any checking required before returning control to the WAIT module. (See also 'The WAIT Statement' in 'PL/I Object Program Management in Multitasking'.)

From the arguments, the interface module is able to determine fully the operation requested of it. The location of the required interface module is available to IHEION from the FCB associated with the file; the field TACM in the FCB is set during the open process to point to the appropriate dynamically loaded module.

Thus, when extra interface modules are provided, the only change required in the open modules is the provision of code to set TACM and any other FCB fields relevant to the new access method interface.

RECORD-ORIENTED I/O CONTROL BLOCKS

Record Dope Vector (RDV)

The record dope vector is an eight-byte block that describes the record variable. Its format depends on the type of statement and the associated options:

Bytes 0-3:	A(INTO/FROM area), or A(POINTER variable) for SET option in READ statement, or A(buffer) for LOCATE statement
Byte 4:	Reserved
Bytes 5-7:	Length of variable

String Dope Vector (SDV)

The address of the string dope vector is passed instead of that of the record dope vector to record I/O interface modules when the input or output of varying strings is requested. The string dope vector is an eight-byte block:

Bytes 0-3:	A(INTO/FROM string)
Bytes 4-5:	Maximum length of string
Bytes 6-7:	Current length of string (output), undefined (input)

Request Control Block

This eight-byte block contains the request codes, in the first four bytes, for various RECORD I/O operations and options. The format is defined in the BREQ field of the I/O control block (IOCB). (See Appendix I.)

The additional four bytes which are contained in the compiler argument list are not copied into the IOCB. Each type of Record-oriented I/O statement is represented by one bit as follows:

Bit number	Statement + options
0	READ SET
1	READ SET KEYTO
2	READ SET KEY
3	READ INTO
4	READ INTO KEYTO
5	READ INTO KEY
6	READ INTO KEY NOLOCK
7	READ IGNORE
8	READ INTO EVENT
9	READ INTO KEYTO EVENT
10	READ INTO KEY EVENT
11	READ INTO KEY NOLOCK EVENT
12	READ IGNORE EVENT
13	WRITE FROM
14	WRITE FROM KEYFROM
15	WRITE FROM EVENT
16	WRITE FROM KEYFROM EVENT
17	REWRITE
18	REWRITE FROM
19	REWRITE FROM KEY
20	REWRITE FROM EVENT
21	REWRITE FROM KEY EVENT
22	LOCATE SET
23	LOCATE SET KEYFROM
24	DELETE
25	DELETE KEY
26	DELETE EVENT
27	DELETE KEY EVENT
28	UNLOCK KEY
29-31	Reserved

I/O Control Block (IOCB)

Record-oriented I/O employs several data management access methods that require that operation requests be provided with a special form of parameter list. This parameter list is termed the data event control block (DECB). A DECB must be provided for each operation, but may be reused when the operation is completed. If several operations are outstanding (owing to the use of the EVENT option in I/O statements, or multitasking), then one DECB is required for each operation.

In order to meet these requirements, the PL/I open process allocates one or more I/O control blocks (IOCB), which are subsequently manipulated or increased in number as follows:

DIRECT access (BISAM and BDAM):
The IOCBs are created by IHEITE(BISAM) or IHEITF(BDAM); for multitasking, they are created by IHEITH(BISAM) or IHEITJ(BDAM). Only one IOCB is created at open time; any others required are created when needed.

SEQUENTIAL access (BSAM only):
All the required IOCBs are obtained at open time; an attempt to use more than those already in existence raises the ERROR condition.

The IOCB format for both these usages is described in Appendix I.

A number of IOCB fields exist in order to support the EVENT option. Since the operation is split into two parts -- initiation through the READ, WRITE, etc., statements, and completion by the WAIT statement -- information regarding a particular operation must be retained for use at the time of completion. For example, if a hidden buffer is employed for a READ, the address of the user's record variable must be retained for subsequent movement from the buffer to the specified area.

<u>IOCB -- SEQUENTIAL Usage:</u> Manipulation of IOCBs for SEQUENTIAL usage is required only for BSAM, which is employed for:

1. CONSECUTIVE UNBUFFERED files.

2. SEQUENTIAL creation or access of REGIONAL files which have the KEYED attribute or are unbuffered.

A number of IOCBs is allocated during the open process by means of the GETPOOL macro; subsequent selection of a particular IOCB is made by a routine similar to that provided by the GETBUF macro. Whenever an IOCB is selected, it is entered into the chain of IOCBs currently in use; the TLAB field in the FCB points to the last IOCB to be used.

The chain of IOCBs is required for two reasons:

1. All I/O operations must be checked in the order in which they were issued.

2. Detection of dummy records for a REGIONAL (2) or (3) data set requires reordering of outstanding requests (due to the use of the EVENT option).

This chain, however, is principally required for the EVENT option, which can cause more than one I/O operation to be outstanding at a given time.

The number of IOCBs (buffers) allocated is determined by the DD statement subparameter NCP. The value of this subparameter should not be greater than 1 unless the EVENT option is employed; if NCP = 1, there is then one IOCB and one channel program. If NCP is unspecified a default of 1 is used.

The size of each IOCB varies, depending upon the organization, the record format of the data set, and whether or not the file (if REGIONAL) has the KEYED attribute. Figure 66 in Appendix I specifies the size requirements.

IOCB -- DIRECT Usage: Manipulation of IOCBs for DIRECT usage is required for both EDAM and BISAM. One IOCB is allocated to a DIRECT file when it is opened; subsequent selection of an IOCB is performed by the modules IHEITE, IHEITF, IHEITH, and IHEITJ. Unlike SEQUENTIAL access, the order of I/O operation is not normally considered. (However, see the BISAM interface modules IHEITE and IHEITH.)

The chain of IOCBs for a given file is anchored in the TLAB field in the FCB; the chain may be extended beyond the original single IOCB if the EVENT option or multitasking is used. An extension occurs if, while there exists an I/O operation that has not been completed, another I/O operation is initiated.

IOCBs for DIRECT access are obtained in subpool zero, in order to cope with multitask manipulation of the chain. The chain of one or more IOCBs is released when the file is closed.

Exclusive Block

When a DIRECT UPDATE file is opened in a multitasking environment, the interface module IHEITH (BISAM) or IHEITJ (BDAM) is loaded instead of IHEITE or IHEITF. IHEITH and IHEITJ contain code to implement the EXCLUSIVE attribute. When a record is locked, an exclusive block is created in subpool 1 of the current task; the block is freed when the record is unlocked. The exclusive block contains the qname (address of the FCB for the file) and rname (region number for REGIONAL(1), region number and key for REGIONAL(2) and (3), and key for INDEXED) required by the ENQ and DEQ macro instructions that are issued to lock and unlock the record. The format of the exclusive block is given in Appendix I.

ACCESS METHOD INTERFACES

This section describes how the PL/I Library relates to the various data management access methods for record-oriented I/O, and gives details of the support required from the library for various PL/I features. This information supplements, but does not replace, that provided in the module summaries and in the module listing prefaces.

CONSECUTIVE Data Sets

The access methods employed for this organization are QSAM and BSAM. The choice between them is governed by the file attributes BUFFERED and UNBUFFERED:

BUFFERED:	QSAM (All record formats)
UNBUFFERED:	BSAM (F,V,U) (Blocked records are illegal)

QSAM (IHEITG): A BUFFERED file is specified in order to take advantage of automatic transmission, process-time overlap, and blocking or deblocking of records. All record formats may be handled.

The locate mode of the GET and PUT macros is employed with this access method (except for paper tape devices) for the following purposes:

1. To support the SET option in READ and LOCATE statements, and to support the REWRITE statement without the FROM option. Module IHEITG allocates the data management buffers for the records, and sets the pointer appropriately. The first byte of a buffer is always on a doubleword boundary; for blocked records, the user must ensure that his alignment requirements are met by adjusting the lengths of the variables being transmitted.

2. To remove or add V-format control bytes if the INTO or FROM option is employed.

Paper tape input requires the use of the move mode to effect translation of the characters transmitted. The open process establishes a work area, placing its address in TREC; the GET macro instruction specifies this area as the receiving area. If an illegal character is read from the paper tape, the access method (QSAM) passes control to the SYNAD routine in IHEITG; control returns from the SYNAD routine to QSAM. When the GET macro instruction has been satisfied, the data is moved into the record variable or a pointer is set, and the TRANSMIT condition is raised.

Closing a data set being created by QSAM may cause output records to be written by the close executor. If an error occurs during the closing process, the operating system uses the ABEND macro to end the task.

QSAM Spanned Records (IHEITK,IHEITL): Buffered VS- or VBS-format records are processed using QSAM Locate Mode for input (module IHEITK) and QSAM Data Mode for output (module IHEITL).

The methods employed are similar to those described above for module IHEITG although the following should be noted:

1. Update Mode (REWRITE) is not supported by the library, since it is not possible to update complete records (O/S restriction).

2. The use of LOCATE or READ SET statements will cause a work area to be established equal to the maximum record size. This area is only released if there is a subsequent READ (without SET) or WRITE statement.

BSAM (IHEITB): An UNBUFFERED file is specified in order to avoid the space and time overheads of intermediate buffers when transmitting records. Overlap of transmission and processing time is only available if the EVENT option is employed.

BSAM requires the use of DECBs to communicate information regarding each I/O operation requested of it; see 'I/O Control Block (IOCB)' and Appendix I (IOCB) for details of the DECB. IHEITB selects an IOCB (which contains a DECB area) from the IOCB (buffer) pool for each input/output operation. The IOCBs used for CONSECUTIVE organization do not contain hidden buffers, except when V-format records are employed. Hidden buffers are used in this case so that the V-format control bytes can be eliminated from the record before the data is moved into the record variable. If, however, the data set consists of F-format unblocked records, and the size of a record variable is less than the fixed size of data set records, a temporary buffer area is dynamically obtained. The use of a temporary buffer area for input prevents the destruction of data following the INTO area; for output, it prevents triggering of the fetch-protect interrupt.

INDEXED Data Sets

The access methods employed for this organization are QISAM and BISAM; they are used thus:

QISAM: SEQUENTIAL creation and access
BISAM: DIRECT access

All usage of INDEXED data sets requires the presence of buffers, even though the file is UNBUFFERED or DIRECT. The buffer is required in order to deal with a 10-byte overflow record link-field. Only F-format records, blocked or unblocked, are permitted.

QISAM (IHEITD): SEQUENTIAL creation and access of INDEXED data sets is performed using this access method. Creation requires that keys be presented in ascending collating sequence. The sequence is checked by the library before the PUT macro is executed, in order to synchronize a given WRITE statement with the raising of the duplicate KEY condition. This arrangement is necessary because, since PUT LOCATE is employed, QISAM would normally raise the condition only on the subsequent PUT operation.

For records with embedded keys, when a WRITE statement with a KEYFROM string shorter than the key length, or a LOCATE statement, is executed, the KEYFROM string is placed in an area addressed by TPKA in the FCB. In the next operation on the file after a LOCATE statement (including a CLOSE statement), the KEYFROM string is compared with the key embedded in the data in the buffer. If they are unequal, the KEY condition is raised. On normal return from the on-unit, control passes to the next statement in the program (i.e., the one following that which caused the KEY condition to be raised). The process of comparing keys and raising the KEY condition is repeated in successive statements that refer to the file until the embedded key has been changed. (After a LOCATE statement has been executed, no further operations are possible on the file until the record has been transmitted; for records with embedded keys, this cannot occur until the KEYFROM string matches the embedded key.)

When a file is closed implicitly (i.e., on termination of a task), the KEYFROM string overwrites the key part of the record in the buffer, and the record is written onto the data set. If the KEYFROM string is not identical with the embedded key, a message is printed out at the console.

To support the REWRITE statement without the FROM option, the key is saved on execution of a READ statement with the SET option. When the REWRITE statement is executed, if the embedded key is the same as the saved key, a PUTX macro instruction is issued. If the key has changed, the PUTX macro is not issued and the KEY (specification) condition is raised.

To support the DELETE statement without the KEY option, the first byte of the logical record is set to X'FF' and a PUTX macro instruction is issued to rewrite the record.

If the file has the KEYED attribute, and the mode is INPUT or UPDATE, the QISAM SETL function is required in order to reposition the indexes. The parameters for the SETL macro are such that, for unblocked records, the recorded key is transmitted as well as the data record. For a READ statement, if the KEY string is shorter than the key length, the string is placed in an area addressed by TPKA in the FCB. If the file is not KEYED (indicating that the KEY option will not be employed), the QISAM SETL routine is not loaded during the open process.

Since buffers are employed, truncation or padding of records is performed during the move between the buffer and the record variable. Padding bytes are undefined in value.

Closing a data set being created or updated by QISAM may cause output records to be written. If an error occurs, output entry to the SYNAD routine is prevented by the close process having cleared the DCBSYNAD field before issuing the CLOSE macro. The operating system uses the ABEND macro to terminate the task.

<u>BISAM in a Non-Multitasking Environment (IHEITE):</u> When the TASK option is not employed, direct access of INDEXED files, both exclusive and non-exclusive, is performed by module IHEITE. For an exclusive file, IHEIOG treats the UNLOCK statement as 'no operation' (although it may implicitly cause the file to be opened); the NOLOCK option is ignored by IHEITE.

BISAM requires the use of DECBs to communicate information regarding each I/O operation requested of it; see 'I/O Control Block (IOCB)' for details of the DECB and its use in BISAM.

Since the EVENT option may be employed, and, moreover, the KEYFROM or KEY expression may yield a character-string value in temporary storage, the key value is moved into the buffer before BISAM is invoked. Truncation or padding of the character-string key to conform to the KEYLEN specification is performed during the move. A further reason for the move is that BISAM may destroy the contents of the key and record fields when adding new records to a data set.

If the data set consists of unblocked records, a READ statement need not precede a REWRITE statement. If blocked records are used, the sequence must be READ, then REWRITE, since the READ macro instruction has the KU parameter, and BISAM requires this type of READ to be rewritten. The WRITE K macro instruction used to rewrite the updated block must address the same DECB(IOCB) as that used for the READ KU macro instruction. This is achieved by not freeing the IOCB used for the READ operation. On the next operation on the file, a check is made for such an IOCB: if one exists, and the operation is not a REWRITE specifying the same key, the ERROR condition is raised.

A DELETE statement is implemented by first issuing a READ KU macro instruction, then setting the first data byte to X'FF', and finally rewriting the record with a WRITE K macro instruction.

<u>BISAM in a Multitasking Environment (IHEITH):</u> To ensure that the initialization and chaining of event variables, IOCBs, and exclusive blocks cannot be interrupted, the interface module IHEIOG raises the dispatching priority of the current task to its limit before calling IHEITH. IHEITH restores the priority to its original value before executing an I/O macro instruction. The formats of the event variable and the exclusive block are described in Appendix I, which also includes an example of the chaining of these blocks.

For non-exclusive files, module IHEITH performs the same functions as IHEITE, and in addition chains any event variables that are made active. Each event variable is placed in a chain anchored in the pseudo-register IHEQEVT in the PRV for the current task. This chain enables I/O event variables for which a WAIT statement has not been executed to be set complete, inactive, and abnormal when the task is terminated.

The implementation for exclusive files includes the following additional features:

1. <u>Files with unblocked records</u>: When any operation referring to a record (except WRITE and UNLOCK) is initiated, the chain of exclusive blocks anchored in the TXLV field of the FCB is searched for an existing exclusive block established in the current task

for the record. If one exists, the lock statement count (XSTC) in the exclusive block is incremented by one. If there is no exclusive block, one is created in subpool 1 and inserted in the task chain (anchored in pseudo-register IHEQXLV in the current task) and the file chain (anchored in the TXLV field of the FCB of the current file). The lock statement count is set to one, and the lock bit (XLOK) to one (unless the operation is READ with NOLOCK), and the resource is enqueued (i.e. the record is locked). After control of the resource has been obtained, it is dequeued if XLOK = 0. The qname and rname given in the ENQ and DEQ macro instructions are:

```
qname (two words):
   Byte 0:      Zero
   Bytes 1-3:   A(FCB)
   Bytes 4-7:   Zero

rname (one word):
   Byte 0:      X'03
   Bytes 1-3:   A(FCB)
```

After the CHECK macro instruction for the I/O operation has been executed (i.e., on execution of the WAIT statement if the EVENT option is used), IHEITH raises the priority of the current task to its limit, decreases the lock statement count by one, and then:

1. If the record is no longer locked (XLOK=0) and the lock statement count is zero, dechains and frees the exclusive block.

2. If the record is still locked (XLOK=1), unlocks it (unless the statement is READ without the NOLOCK option), and sets XLOK to zero. If the lock statement count is zero, it then dechains and frees the exclusive block.

IHEITH then restores the dispatching priority to its original value.

2. Files with blocked records: To prevent other tasks interfering with the READ, REWRITE sequence, each READ, WRITE, REWRITE, and DELETE statement is enqueued on the same resource (i.e., there is only one exclusive block for each file in each task, and it is not freed until the file is closed). Control of the resource is retained by a given task until the WRITE, REWRITE, or DELETE operation is completed; or, if the resource was enqueued by a READ operation, until a REWRITE or UNLOCK statement is executed. When a READ statement with the NOLOCK option is executed, the resource is dequeued immediately after the task gains control of it.

   Apart from these differences, the implementation is as for files with unblocked records.

REGIONAL Data Sets

The access methods employed for these organizations are BSAM and BDAM, as follows:

BSAM: Creation and SEQUENTIAL access
BDAM: DIRECT access

Keys supplied by the source code are termed 'source keys'. These have two formats, one of which is interpreted in two ways:

Organization	Source key format
REGIONAL (1): Relative record addressing, without recorded keys	A
REGIONAL (2): Relative record addressing, with recorded keys	B
REGIONAL (3): Relative track addressing, with recorded keys	B

Key Format A:

```
┌───────────────────────────────┐
|               M               |
└───────────────────────────────┘
<---------------L--------------->
```

L = Length of key (1 through 255 bytes)
M = Key value

Only the characters blank and 0 to 9 may be used in M, which, when converted to binary, is the relative record position, as required for the BDAM BLKREF parameter. The last eight characters are scanned for an unsigned decimal integer representation; if less than eight characters exist, only the available characters are scanned, from left to right.

When a format-A source key is required for the KEYTO option, the relative record position of the current record is converted from a binary count field into character representation and is assigned to the last eight characters of the KEYTO character string variable. If the variable has fewer than eight characters, the converted value is assigned, right to left, to the KEYTO variable. Format A keys are not appended to data set records as recorded keys.

Key Format B:

```
┌──────────────┬──────────────┐
│      C       │      M       │
└──────────────┴──────────────┘
<──────────────L──────────────>
```

L = Length of key (1 through 255 bytes)
M = Last eight characters in the source key
C = The remaining characters in the source key other than the M characters

M consists of up to 8 characters, which can be blanks or 0 to 9. When converted to binary, it represents either the relative record position (REGIONAL (2)), or the relative track position (REGIONAL (3)).

If L ≤ 8, C does not exist. The C characters can be any of the 256 characters available; they are not scanned.

The format-B source key is appended to output records when they are added to the data set; the number of characters in the appended (recorded) key is determined by the KEYLEN specified for the data set. If KEYLEN is less than the length of the source key, the latter is truncated when appended to its record; if greater, the source key is padded with blanks. Similarly, when retrieving keyed records, the source key is altered to conform to KEYLEN. This permits 1 though L characters to be used as the recorded key. The M characters might thus be used only for computation of the relative record or track position.

BSAM (IHEOPZ, IHEITC, IHEITB): Creation and sequential access of REGIONAL data sets employs this access method.

SEQUENTIAL creation is performed by the module IHEITC, which adds records to the data set in physically sequential record and track positions. This module also inserts dummy records, as required, by the user incrementing the source key position information by a value greater than one.

When a sequentially created REGIONAL data set is closed, the current space allocation (which may be either the initial cr a secondary allocation) is completed:

1. by writing dummy records (F-format only), or

2. by setting the capacity records of the remaining tracks to indicate empty tracks.

An FCB history flag (TMET) is turned on when, after writing a record, this record is seen to be the last one of an extent. If this flag is off, the close process will continue the initialization until an end-of-extent condition is met.

When LOCATE statements are used to create a REGIONAL data set, an IOCB is selected from the pool in the normal manner. The KEYFROM string is evaluated, and all necessary formatting of the data set is done, before the pointer is set and control is returned to compiled code. To ensure that the record is always aligned on a doubleword boundary, the open process rounds up the keylength to a doubleword and allows space in the IOCB for the keylength and the block size. Module IHEITC places the key right-aligned in the key area, thus ensuring that the key and data are in contiguous areas, and that the data is aligned on a doubleword boundary.

The record is not actually transmitted until the next statement on the file (e.g., CLOSE, WRITE, LOCATE) is executed. If it is found on transmission that there is no room for the record in the region (REGIONAL(3) V and U format records only), the capacity record is written and the KEY sequence error condition is raised. On normal return from the on-unit, control passes to the next statement. If this occurs when a file is closed implicitly (on termination of a task) or explicitly, a warning message is printed and the file is closed (after the initialization of the current extent has been completed). Note that it is therefore possible that the original record associated with the LOCATE statement may not have been written.

DIRECT creation requires the initialization of the data set during the open process; this is performed by the module IHEOPZ. Subsequently, records may be added to the data set in a DIRECT fashion using module IHEITF or IHEITJ. Initialization of a data set for DIRECT creation causes:

1. the initial space allocation (secondary allocation is ignored) to be written with dummy records (F-format records, for all REGIONAL types), or

2. the capacity record of each track of the initial space allocation to be set to indicate empty tracks (U-format or V-format records, REGIONAL (3), only).

If recorded keys are required, dummy keys (initial byte X'FF', remaining bytes undefined) are also written for F-format records only. If during the initialization for DIRECT creation an error arises, the UNDEFINEDFILE condition is raised, the type of error being indicated by the ONCODE value.

As SEQUENTIAL access of a REGIONAL data set (module IHEITB) is performed with BSAM, it is not possible to support the KEY option on the READ statement. The KEYTO option is supported as follows:

REGIONAL (1):
A counter (the TREL field in the FCB) beginning at zero, is incremented as each record, including dummy or deleted records, is read; this is converted to character string representation and assigned to the KEYTO variable.

REGIONAL (2) and (3):
The recorded key is read in with the record, and assigned, without conversion, to the KEYTO variable. Transmission of the recorded key only occurs if the file has the KEYED attribute; otherwise the KEYLEN DCB field is forced to zero to prevent input of keys (since, for F or U records, there are no hidden buffers).

For both SEQUENTIAL creation and access, BSAM requires the use of DECBs to communicate information regarding each I/O operation requested of it; see 'The I/O Control Block (IOCB)' for details of the DECB and its use for BSAM. When REGIONAL data sets with the UNBUFFERED attribute are accessed (IHEITB) or created (IHEITC), hidden buffers are present in all cases except for REGIONAL(1), since the key and data must be within a contiguous area in a buffer.

When reading REGIONAL data sets sequentially, BSAM retrieves all records within the data set, whether dummy (deleted) or actual records. For REGIONAL (2) and (3) data sets, the library prevents dummy (deleted) records being passed to the PL/I program. This is achieved by inspecting the initial byte of the recorded key as transmitted to the hidden buffer. (Hidden buffers are always required for KEYED SEQUENTIAL access of REGIONAL (2) and (3) data sets, because BSAM requires that the recorded key and the record be transmitted into contiguous storage areas.)

If the initial byte is the dummy, or deleted, code (X'FF'), the IOCB chain is reorganized to move each input request down one entry in the chain; this resynchronizes the READ statements with the actual records. The reorganization occurs each time such a flagged key is detected. This technique is not available for REGIONAL (1), since for this type of organization:

1. there is no way of knowing whether the records are actual or dummy, since there are no restrictions regarding the initial byte of the data record, and

2. there are no recorded keys.

When a READ statement with the SET option is executed for REGIONAL files, the data is always aligned on a doubleword boundary in the IOCB buffer.

<u>BDAM(IHEITF and IHEITJ):</u> DIRECT access to a REGIONAL data set employs this access method, the usage depending upon the REGIONAL type:

REGIONAL (1):
Relative record (block) addressing, no key argument

REGIONAL (2):
Relative record (block) addressing, with key search argument

REGIONAL (3):
Relative track addressing, with key search argument

In the instance of REGIONAL (2) and (3), the "extended search" feature is always employed. A user may control the effects of extended search by using the DCB subparameter LIMCT; a value may be specified to limit the number of records or tracks which are searched for a given keyed record, or for space to add one. Unless so limited, searching for records extends throughout the complete data set.

The BDAM access method requires the use of DECBs to communicate information regarding each I/O operation requested of it; see 'I/O Control Block (IOCB)' for details of the DECB and its usage for BDAM. If V format records are used, any IOCB created will contain a hidden buffer.

The BDAM CHECK macro is issued to check that the operation is complete. If an error is found, the BDAM modules enter the IHEITF SYNAD routine, where the error is interrogated.

If the TASK option is not used, direct access of REGIONAL files, both exclusive and non-exclusive , is performed by module | IHEITF. For an exclusive file, IHEION treats the UNLOCK statement as 'no operation' (although it may implicitly cause the file to be opened); the NOLOCK option is ignored by IHEITF.

If the TASK option is employed, module IHEITJ is loaded instead of IHEITF. The difference between these modules is the same as that between IHEITE and IHEITH for unblocked records. (See 'BISAM in a Multi-tasking Environment'.)

INTRODUCTION

The PL/I Library provides facilities for the dynamic management of PL/I programs. This involves:

1. Program management: Housekeeping at the beginning and end of a program or at entry to and exit from a block.

2. Storage management: Allocation and freeing of storage for automatic and controlled variables, and for list processing.

This section describes the requirements for these facilities and their implementation by the library. With the exceptions of the compiler optimization routine and storage management for list processing, all the functions described are performed by module IHESA, whose entry points are listed in Figure 16; full details are given in Chapter 9. Object program management in a multitasking environment is discussed in Chapter 5.

Entry point	Function
IHESADA	Get DSA
IHESADB	Get VDA
IHESADD	Get controlled variable
IHESADE	Get LWS
IHESADF	Get library VDA
IHESAFA	END
IHESAFB	RETURN
IHESAFC	GO TO
IHESAFD	Free VDA/Free LWS
IHESAFF	Free controlled variable
IHESAFQ	Abnormal program termination
IHESAPA IHESAPB IHESAPC IHESAPD	Program initialization
IHESARA	Environment modification
IHESARC	Setting of return code

Figure 16. IHESA Entry Points

Program Initialization

Certain functions must be carried out on entry to a PL/I program before the PL/I main procedure is given control. One of the library program-initialization subroutines is always given control by the supervisor on entry to the program. Its functions are:

1. Allocation of storage for the PRV. (See 'Communications Conventions' in Chapter 2.)

2. Initial allocation of LWS.

3. Passing of the address of the library error-handling subroutine (IHEERR), which assumes control when an interrupt occurs, to the supervisor.

Block Housekeeping: Prologues and Epilogues

Prologues and epilogues are the routines executed on entry to and exit from a PL/I procedure or begin block. The library subroutines contain those sections that are common to all prologues and epilogues. The functions of the library prologue subroutine are:

1. To preserve the environment of the invoking block.

2. To obtain and initialize automatic storage for the block.

3. To provide chaining mechanisms to enable the progress of the program to be traced. A detailed description of the chaining mechanisms employed is provided below.

The main functions of the epilogue subroutine are:

1. To release storage for the block.

2. To recover the environment of the invoking block before returning control to it.

Storage Management

In IBM System/360 Operating System, storage is obtained or freed by using the GETMAIN and FREEMAIN macros. The library assumes responsibility for obtaining and freeing storage in this way in order to:

1. Provide an interface between compiled code and the control program.

2. Reduce the overhead involved in making a supervisor call every time storage is obtained and freed.

3. Set up chaining mechanisms for dynamic storage.

There are three types of dynamic storage in PL/I, controlled, automatic, and based. Based storage is discussed in 'List Processing: Storage Management'.

Operating-System Facilities

The following facilities appropriate to this chapter are provided by IBM System/360 Operating System. (See IBM System/360 Operating System: Supervisor and Data Management Macro Instructions.)

SPIE macro instruction: Specifies the address of a routine to be entered when specified program interrupts occur.

ABEND macro instruction: Causes a job step or task to be terminated abnormally.

Write To Operator (WTO) macro instruction: Can be used to write a message on the operator's console.

R-type GETMAIN: Requests that the supervisor allocate a contiguous block of main storage to the caller. A subpool number should be specified. (See below.)

R-type FREEMAIN: Releases a main storage area. The length, subpool number, and address of the beginning of the area must be specified.

Subpools: Subpool numbers are of significance only in an operating system with MVT.

Subpool zero
The storage in subpool zero is allocated on a job-step basis, and is never automatically released until the end of the job step.

Subpool non-zero
The storage in a subpool with a non-zero number is allocated on a task basis, and is automatically released on the termination of the task that owned the subpool.

IBM System/360 Operating System: Supervisor and Data Management Services contains a full discussion of main-storage management.

AUTOMATIC STORAGE: STORAGE MANAGEMENT

Two types of automatic storage area are needed to implement the functions described above. These are:

1. The storage area associated with the execution of a PL/I block, known as a dynamic storage area (DSA).

2. The storage area mainly used for automatic variables whose extents are unknown at compile time, known as a variable data area (VDA).

Each type of storage area is identified by flags set in the first byte. These flags also indicate the existence of certain optional entries in the storage area. The flag patterns are shown in Appendix J.

Dynamic Storage Area (DSA)

This area, always associated with the execution of a PL/I block, is used to record the progress and environment of a program. It also contains space for AUTOMATIC variables declared in the block and for various optional entries. The minimum size of a DSA is 100 bytes. The format is described in Appendix J.

The address of the DSA associated with a particular block is held in a pseudo-register. Hence there is a pseudo-register for each block; the group of these pseudo-registers is known as the display. The address contained in a display pseudo-register can be used to identify the DSA associated with a non-recursive block when a GO TO statement specifying a label in that block is executed.

When a block is entered recursively, a new DSA is created for the invoked block. The address of the DSA associated with the previous invocation of that block is stored in the display field of the new DSA. This address is already stored in the appropriate pseudo-register, where it is now replaced by the address of the new DSA. When this latest invocation is finished, the new DSA is freed and the address of the previous DSA is restored to the appropriate pseudo-register.

When there is a GO TO statement to a label in a recursive block or to a label variable, a unique means of identifying the block containing the label is needed. This is accomplished by means of an invocation count, which is stored in the invocation-count field in the DSA during the prologue. The current invocation count is contained in a pseudo-register and is increased by one each time a DSA is obtained.

Variable Data Area (VDA)

A variable data area is a special type of automatic storage area used for variables whose extents are not known at compile time. This storage area is associated with the storage obtained for a particular block. The only housekeeping necessary is that which provides a means of identification of the type of storage area and a method of associating it with a particular block for epilogue purposes.

VDAs are used for three other purposes:

1. Temporary storage for library modules. These areas are only distinguishable from an ordinary VDA by the flag byte. This is to allow them to be freed on a GO TO, as described in the example in 'DSA Chain' under 'Block Housekeeping'.

2. The PRV and primary LWS are contained in a VDA known as the PRV VDA which is chained back to the external save area.

3. Secondary LWS is contained in a special library workspace VDA.

The formats of the VDA, PRV VDA, and LWS VDA are shown in Appendix J.

Library Workspace (LWS)

The housekeeping associated with library workspace can be divided into two parts:

1. The identification of the area needed as library workspace, and chaining this to a previous allocation of automatic storage and to any previous library workspace.

2. The updating of the pseudo-registers pointing at the various areas in library workspace.

The first allocation of LWS is contained in the PRV VDA; subsequent allocations are contained in the LWS VDA. The pseudo-register IHEQLSA always contains the address of the current LWS. Save areas within LWS are indicated thus:

1. The address of each save area is held in a pseudo-register.

2. The beginning of each save area is indicated by X'60' in the first byte. (A DSA can often be readily distinguished from a save area in LWS by the presence of X'8' to X'F' in its first half byte. Appendix J includes the format of the first byte of the DSA.)

Allocation and Freeing of Automatic Storage

This section describes the methods of controlling the allocation and freeing of automatic storage for VDAs, DSAs and secondary LWS.

To minimize the number of supervisor calls necessary to obtain automatic storage, a fairly large block of storage is obtained every time a call is made. Areas are allocated by the library from this block as required until a request is made that is too big to be satisfied from the remaining storage in the block. Another block is then obtained by a call to the supervisor. So that a check can be made as to whether the amount of storage remaining in a block is sufficient to meet an allocation, a record of the amount is stored in the block. When a storage area is freed, its length is added to the available length in the block. When the available length equals the total length of the block, the block is returned to the supervisor.

Since storage areas are released in the reverse order to their allocation, a chain-back mechanism, with a pointer to the last member of the chain, is provided.

Initially, storage is allocated for the PRV VDA from a 4k or a 6k block. When further requests are made for storage, they are satisfied by allocations from the remaining storage of this block. When a request cannot be satisfied, a 2k block (or a block containing a multiple of 2k bytes) is obtained by means of a GETMAIN macro. This block is chained to the existing block by the free-core chain. (See Figure 17.)

In any block that contains unallocated storage (that is, contains free core), the first four words of the unallocated storage are used for control purposes:

<u>1st word</u>:	Length (in bytes) of the unallocated storage for that block (excluding the four control words)
<u>2nd word</u>:	Block length
<u>3rd word</u>:	A(Free core length in previous block)
<u>4th word</u>:	A(Free core length of following block)

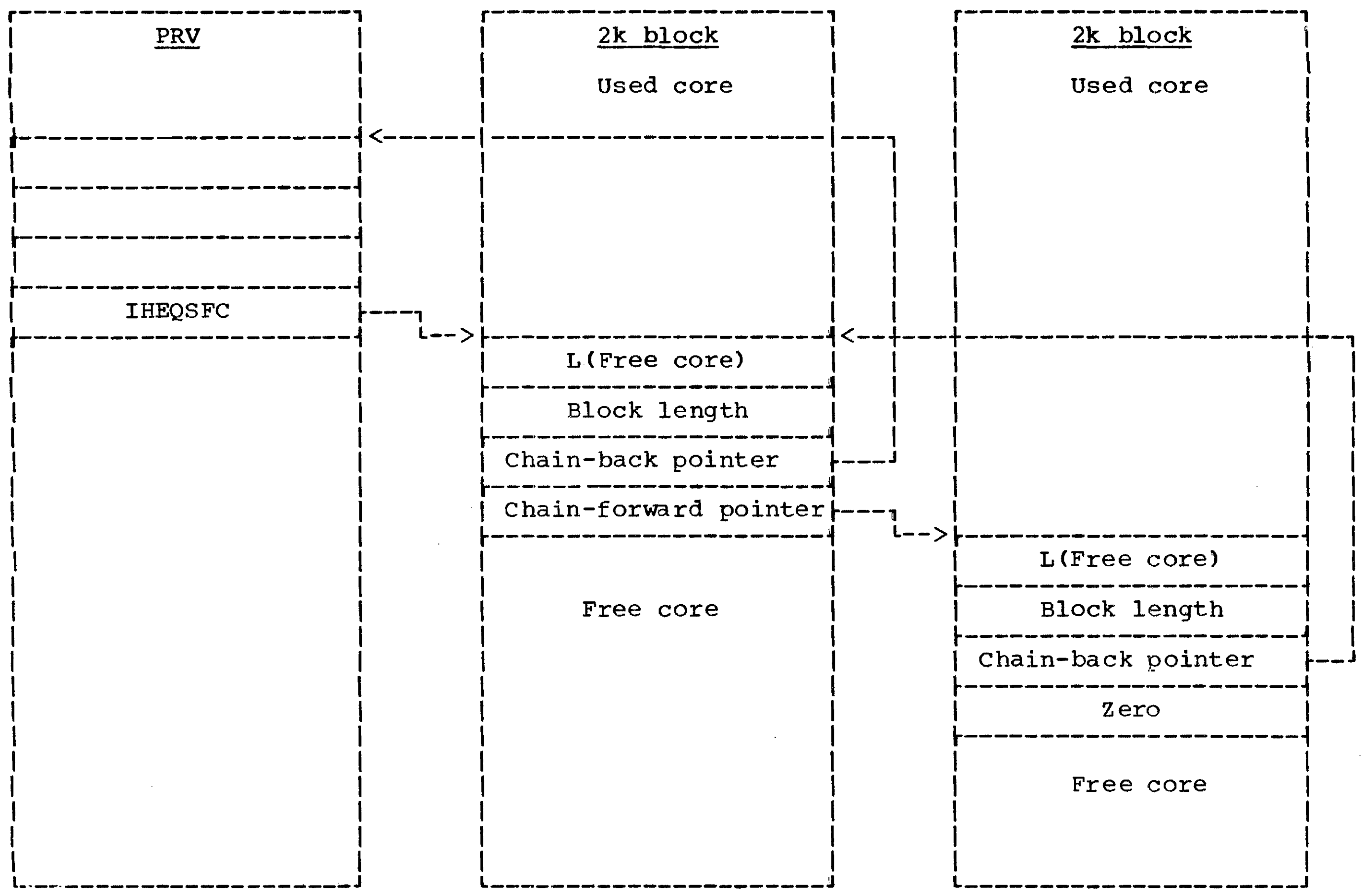


Figure 17. Structure of the Free-Core Chain for Automatic Variables

The first and last blocks require a slightly different usage:

First block: Uses the free-core pseudo-register IHEQSFC in the chaining forward and back:

1. IHEQSFC contains A(Free-core length of first block).

2. 3rd word of block contains (A(IHEQSFC) - 12), which is a dummy free-core length in the PRV.

Last block: 4th word contains 0

When a request for storage is received, a search of the free-core lengths, starting from the first, is made. If a free-core length equal to or greater than the length requested is found, the request is satisfied from that block. The free-core length and pointers are adjusted, as are the appropriate pointers in the blocks on either side.

When storage is freed, the pointers are adjusted, and the free-core field in the corresponding block is updated. If a 2k block becomes available, it is freed by issuing a FREEMAIN macro, and the free-core chain pointers are adjusted accordingly.

CONTROLLED STORAGE: STORAGE MANAGEMENT

Controlled storage is used for controlled variables only; it is requested by the ALLOCATE statement and freed by the FREE statement.

Allocation of a particular controlled variable may occur a number of times. Since the latest allocation is always the one to be used it is convenient to have a pseudo-register pointing at it; this pseudo-register is sometimes referred to as an 'anchor word'. Each allocation is chained back to the previous allocation so that the pseudo-register can be updated when the current allocation is freed (Figure 18). The length of each allocation

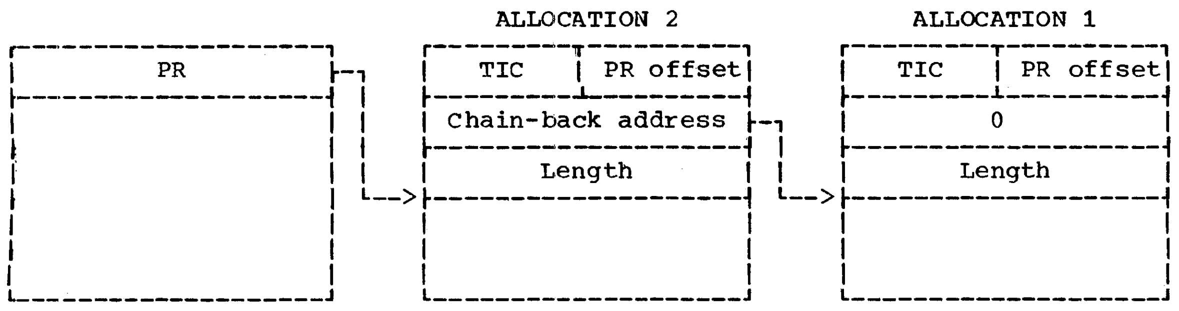


Figure 18. Storage Allocation for a Controlled Variable

is recorded in the fullword field following the chain-back address. The Task Invocation count is held in the TIC field.

When there is no allocation, the contents of the pseudo-register are zero. Each allocation points to the previous allocation, the pointer being zero in the first allocation, which is at the bottom of the stack. Thus the various allocations of a particular controlled variable become part of a push-down (ALLOCATE) pop-up (FREE) list.

When a request is made to storage management for a new allocation, it is serviced by issuing a GETMAIN macro. Twelve bytes are added to the length requested, for control purposes, and this new length is rounded up to a multiple of eight bytes. The length field contains the actual length requested. The pseudo-register is updated and points to word four of the area. When a request is made to storage management to free an allocation, it is serviced by updating the pseudo-register and issuing a FREEMAIN macro.

LIST PROCESSING: STORAGE MANAGEMENT

This section describes the functions of module IHELSP, which controls the allocation and freeing of storage for the PL/I list-processing facility. The functions involved are:

1. Allocation and freeing of system storage for based variables.
2. Allocation and freeing of storage for based variables in programmer-defined areas (area variables).
3. Assignments between area variables.

System Storage for Based Variables

Storage for based variables is allocated and freed in a similar manner to controlled storage, but it is not stacked since each generation is associated with a particular pointer value: reference may be made to any current generation of based storage by associating the appropriate pointer value with the name of the based variable. A request for a new generation of based storage is serviced by issuing a GETMAIN macro, and storage is freed by the FREEMAIN macro. Based storage is allocated only in multiples of eight bytes: the sum of the length of the variable and its offset from a doubleword boundary is rounded up to a multiple of eight bytes. All based storage allocated in a task is freed at the end of the task.

The AREA Attribute

The AREA attribute enables a programmer to define a block of storage (an area variable) in which he can collect and make reference to based data. Space within the area variable is requested and released by ALLOCATE and FREE statements that include an IN(area-variable) clause. Reference can be made to a based variable contained by an area variable just as if the based variable were in system storage. The contents of one area variable can be assigned to another area variable, and an area variable can be handled as a single data item in input/output operations.

The Area Variable

The format of the area variable is shown in Figure 19. The start of the area is aligned on a doubleword boundary. The first four fullwords are used for control information, the remainder of the area

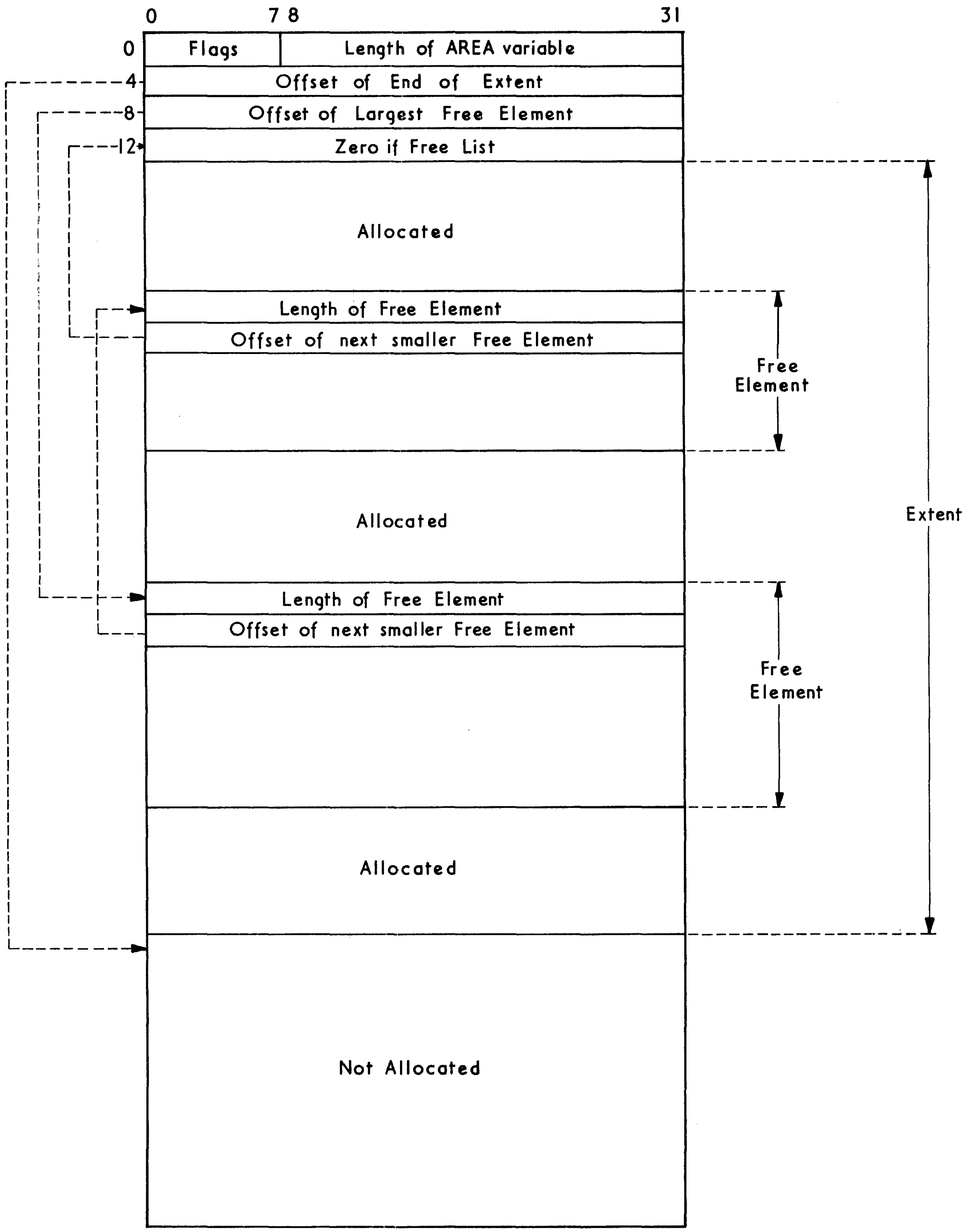


Figure 19. Format of Area Variable

being the storage requested by the programmer in declaring the area variable. The portion of the area that has been allocated to based variables is termed the extent. When storage is allocated to an area variable, its length is set in the last three bytes of the first word, and the second word (offset of end of extent) is set to zero.

Area Storage for Based Variables

Storage for based variables within an area variable is allocated only in multiples of eight bytes; each such allocation is termed an element. The first request for storage for a based variable is satisfied by the allocation of the appropriate number of bytes starting at the beginning of the unused space; the offset of the end of this allocation is set in the second word of the area variable, which now points to the first available doubleword of unused storage. Providing no storage has been freed, further requests are met by further contiguous allocations from the unused space, the offset of the end of the extent being updated each time.

If the last allocation of the extent is freed, the offset in the second word of the AREA variable is reduced. However, if allocations other than the last in the extent are freed, the extent is not reduced: spaces, termed free elements, are left. The length of each free element is set in its first fullword, and a pointer to the next smaller free element (in the form of an offset from the start of the area variable) is set in the second word. If there are no smaller free elements, the second word of the free element points to the fourth word of the area variable, which is set to zero. The chain of free elements is termed the free list, and is anchored in the third word of the area variable, which contains the offset of the largest free element. When an area variable contains a free list, the first bit of the flag byte is set to 1.

Whenever storage in an area variable is to be allocated to a based variable, the free list is searched for the smallest element that will contain the based variable. If no free element is large enough, space is allocated from the unused part of the area. If this, also, is too small, the AREA condition is raised. When an element is freed, it is placed in the free list according to its size. If it is contiguous with another free element, the two are merged and included in the free list as a single element. If the last element in the extent is freed, the extent is reduced and the element is not placed in the free list.

Assignment Between Area Variables

When the contents of area variable A are assigned to area variable B, the current extent and the control words (except the length of A) are copied into B. If the length of B is less than the extent of A, the AREA condition is raised.

The AREA Condition

If an on-unit is entered when the AREA condition is raised during the execution of an ALLOCATE statement, the ALLOCATE statement is executed again after the on-unit has been terminated normally. The return address passed by compiled code is stored in the library communications area (WREA) before the on-unit is entered. On normal termination of the on-unit, IHEERR returns control to the address in WREA.

If the AREA condition is raised during the execution of an assignment statement, the statement is not executed again.

PROGRAM MANAGEMENT

Initialization of a PL/I Program

On entry to a PL/I program, one of the library initialization subroutines (IHESAPA, IHESAPB, IHESAPC, and IHESAPD) is always given control by the supervisor; the entry point that is used depends on the level of compiler optimization required (see below) and on whether the PL/I program is called from an assembler-language routine. The initialization routine first obtains storage for the PRV VDA. The length required is the sum of:

L(PRV) (passed by the linkage editor)

L(LWS) (assembled by the initialization subroutine)

8 control bytes

Since a pseudo-register is referenced by the addition of a fixed displacement to the base address in register PR, and the maximum displacement allowed by the assembler is 4096 bytes, the length of the PRV is limited to 4096 bytes. This puts the upper limit on the combined number of blocks, files and controlled variables at about 1000. If the initialization routine is asked to get a PRV longer than 4096 bytes, a message is printed out on the console and the program is terminated.

The initialization routine zeros the PRV, sets up the LWS pseudo-registers, and issues a SPIE macro instruction naming IHEERR. In addition, IHESAPA and IHESAPC enable a PARM parameter on the EXEC card to be passed to the PL/I program. (See IBM System/360 Operating System: Job Control Language.) On exit from the initialization subroutine, register RA points at a location containing the address of the SDV of the parameter.

Termination of a PL/I Program

Normal Termination: Normal termination of a PL/I procedure is achieved by an END or RETURN statement, either of which involves releasing the automatic storage associated with the procedure. If a request is made to free a DSA which would entail freeing the DSA for the main procedure, IHESAFA (END) or IHESAFB (RETURN) raises the FINISH condition and the program branches to the error-handling subroutine (IHEERR). If and when this subroutine returns control, IHESAFA or IHESAFB causes all opened files to be closed (by calling the library implicit-close subroutine). Subsequently all automatic storage, including the PRV VDA, is returned to the supervisor. IHESARC is then called to set the return code and return control to the supervisor.

Abnormal Termination: A PL/I program is considered to terminate abnormally when the FINISH condition is raised by any means other than a RETURN, END, or SIGNAL FINISH statement (e.g., when an object-time error occurs such that the ERROR condition is raised). If there is not a GO TO out of the ERROR or FINISH on-unit (if any), the error-handling subroutine (IHEERR) calls IHESAFQ, which closes all the open files in the manner described above; IHESAFQ returns to the supervisor with a return code of (2000 + any return code already set (modulo 1024)).

GO TO Statements

In PL/I, a GO TO statement not only involves the transfer of control to a particular label in a block but also requires the termination of contained blocks. The housekeeping requirements for this are:

1. A return address.
2. A means of identifying the automatic storage associated with the block to be made current.

Identification of the appropriate storage depends on whether the environment is recursive or non-recursive:

Recursive: A count (the invocation count) is kept of the number of times any block is entered; this count can be used to identify the storage for a particular invocation.

Non-recursive: The address of the storage for each block is required.

On-Units and Entry-Parameter Procedures

If, in a recursive environment, the program enters:

1. an on-unit, or
2. a procedure obtained by calling an entry parameter,

that environment must be restored to the state that existed when the ON statement was executed or the entry parameter was passed. Similarly, at the exit from the on-unit or the entry-parameter procedure, the environment must be restored to its former state.

If the on-unit or entry-parameter procedure refers to automatic data in encompassing blocks, these references will be to the generations that existed when the ON statement was executed or the entry parameter was passed. These will not necessarily be the latest generations.

The correct environment is obtained by restoring the display to what it was at the time the ON statement was executed or the entry parameter passed.

When an on-unit is to be entered, the library error-handling subroutine calls IHESARA and passes it:

1. The address of the on-unit.
2. The invocation count of the DSA associated with the procedure containing the ON statement.

When an entry-parameter procedure is to be called, compiled code branches to IHESARA and passes it:

. The address of the called procedure.

2. The invocation count of the passing procedure.

The state of the display at the time of

passing is determined by examining the DSAs of active blocks invoked before the passing procedure. The display is modified and control is transferred to the called procedure.

Before an on-unit or an entry-parameter DSA is freed, the display is restored, in a similar manner to that described above, to the state it had immediately before the on-unit was entered or the entry-parameter procedure was called.

<u>Block Housekeeping</u>

The chaining of automatic storage areas is required both for housekeeping purposes and for storage management. In general, both these functions are satisfied by the automatic storage area chain (called the DSA chain or 'run time stack'). When a library module is entered, an offshoot of the DSA chain, known as the save-area chain, may be formed.

<u>DSA Chain:</u> The DSA chain consists of the external save area, PRV VDA, DSAs and VDAs. DSAs are added to the chain as procedures and blocks are entered. VDAs are added to the chain after the DSA of the block in which they are required. The pseudo-register IHEQSLA is always set to point at the last allocation in the chain. Initially it points at the PRV VDA. Register DR always points to the current save area.

Consider a sample program. Successive areas are added to the chain thus:

1. PRV VDA
2. DSA (Main procedure)
3. DSA (Procedure)
4. DSA (Begin block)

At this stage the storage map is as shown in Figure 20. If the begin block required a VDA this would be added to the end of the chain. Figure 21 shows an example in which the begin block required two VDAs. If the program now executes:

1. <u>An END statement</u>: The storage in the chain is released, starting with the area pointed at by IHEQSLA and finishing when the current DSA has been released. This leaves the chain with items 1, 2 and 3 only.
2. <u>A RETURN statement</u>: All areas up to and including the immediately encompassing procedure DSA are released, leaving only items 1 and 2.

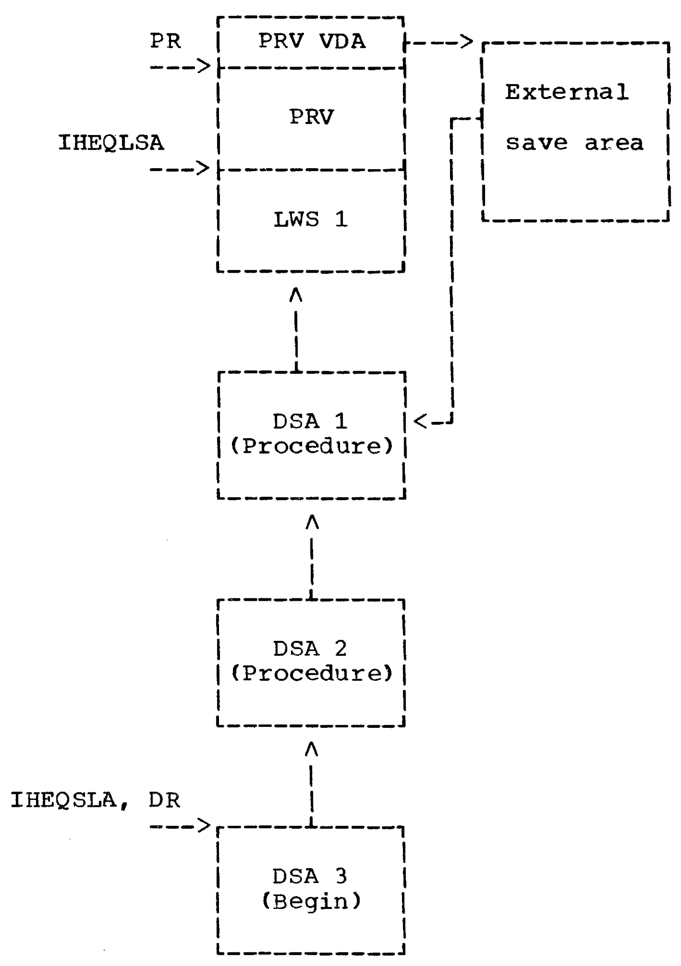


Figure 20. Example of DSA Chain

It is also possible to release the last VDA in a chain without releasing any other areas, by freeing the area pointed at by IHEQSLA.

If a GO TO statement referring to a label in the main procedure had been executed when the situation was as shown in Figure 21, then either the invocation count or the display of the main procedure would be passed to the library subroutine (IHESAFC). This would then search back up the chain until it found the DSA with that invocation count or display, and then make this DSA current. It would then free:

1. All areas up to and including the DSA allocated <u>after</u> the DSA to be made current.
2. Any library VDAs or LWS between the DSA to be made current and the following DSA. A VDA used by the library is distinguished from one used by compiled code by the flags in the first byte. (See Appendix J.)

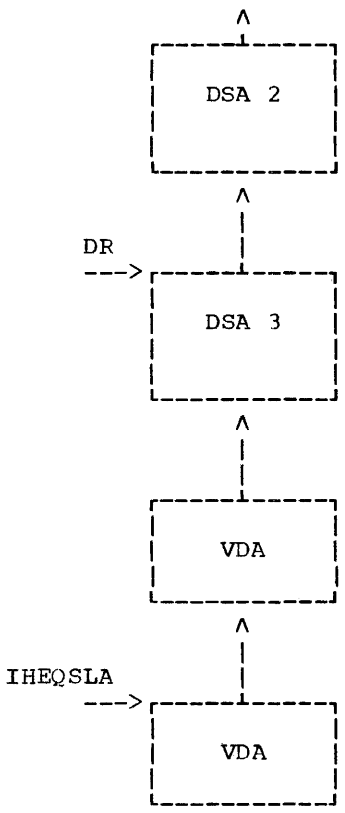


Figure 21. Continuation of the DSA Chain

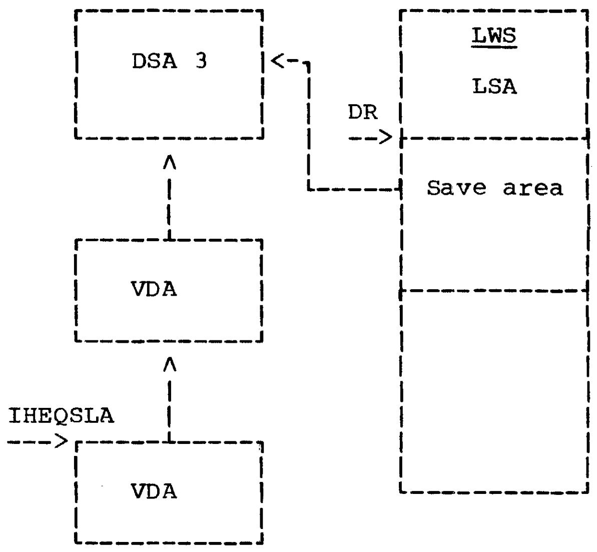


Figure 22. Construction of the Save-area Chain

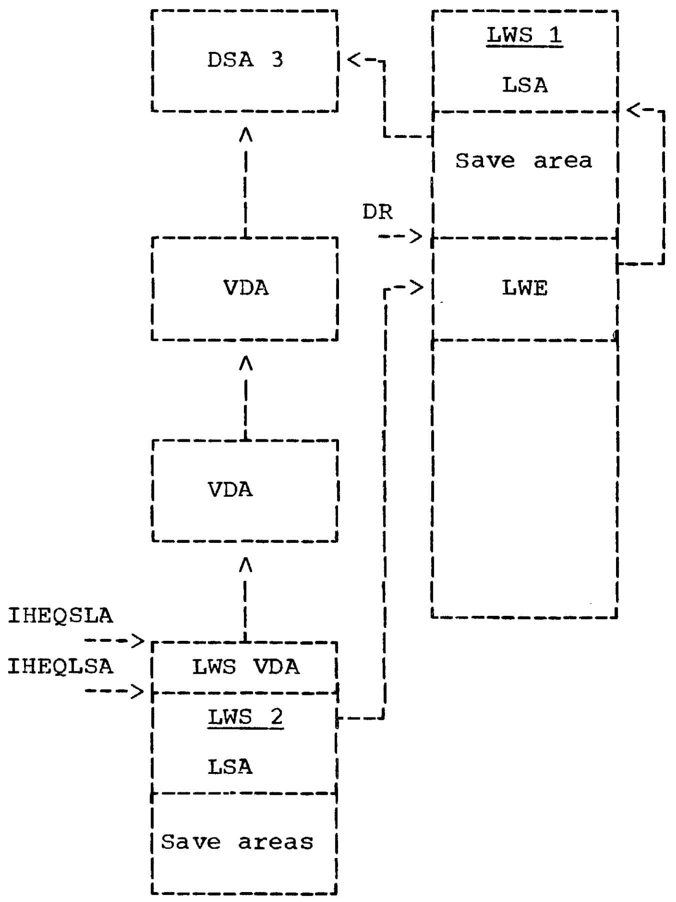


Figure 23. Structure of the DSA chain when the error-handling subroutine is entered after a new LWS has been obtained

Save-Area Chain: When a PL/I block calls a PL/I Library subroutine, the save area passed is that in the DSA for that block. If the library routine calls a lower-level library routine, the save area passed is that of the appropriate level in LWS. Thus a save-area chain is built up as an off-shoot of the DSA chain. (See Figure 22.) Normally the save-area chain unwinds itself as control returns up through the levels; in the example, the chain would be left with DSAs 1, 2 and 3 remaining.

Treatment of Interrupts: When a program interrupt occurs in a subroutine (library or compiled code), the library error-handling subroutine (IHEERR) is entered and the address of the save area of that subroutine is set in register DR. (See Figure 23.)

IHEERR calls IHESADE, passing its own save area, to get a new LWS (LWS2). If there is an on-unit corresponding with the interrupt condition, then, on return from IHESADE, IHEERR branches to IHESARA (which modifies the display) and passes it the save area LSA in LWS2. In turn, IHESARA branches to the on-unit and passes it the same save area. The prologue for the on-unit then calls IHESADA to obtain a DSA. The DSA chain can now continue if required. (See Figure 24.)

If there is no on-unit corresponding to the interrupt condition, standard system action is taken. (See Chapter 6.)

There are two possible ways of freeing the on-unit DSA:

1. By a GO TO statement from the on-unit. If the GO TO is to a statement in a block associated with DSA 3, or earlier, then the save-area chain can simply be forgotten. Registers are restored from the DSA to become current.

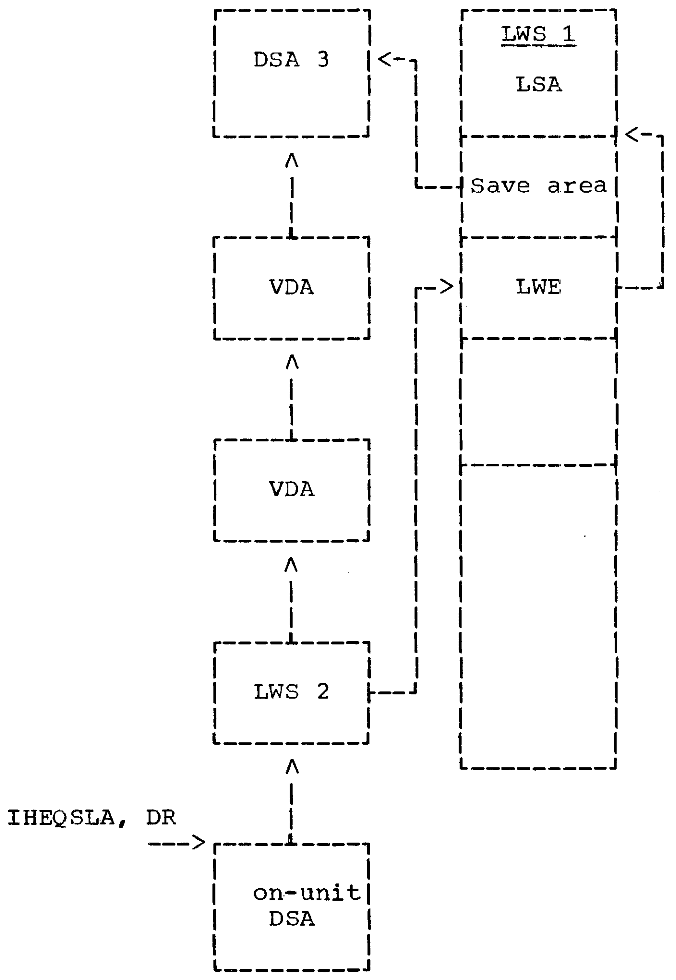


Figure 24. Structure of the DSA chain when the on-unit DSA is attached

2. By the on-unit issuing a request to storage management to free the on-unit DSA. When this is done, control is returned to the error-handling subroutine at the point following that from which control was transferred to the on-unit. The error-handling subroutine restores DR in the normal way to point at LWE in LWS 1 and calls IHESAFD to free LWS 2. Control is then returned to the interrupted routine. In the example, the situation would now be as in Figure 22.

Object-time Optimization

The compiler contains an optimization technique which minimizes the necessary housekeeping and provides faster execution of the prologue and epilogue. The technique can only be applied if the optimization option (OPT=01.Default) is specified for the compilation of the main procedure of a program. In this case, in a non-multitasking environment, a 512-byte storage area is reserved at the end of primary LWS during initialization. The pseudo-register IHEQLWF contains the address of the reserved area attached to the current LWS. A reserved area is released only when its associated LWS is released.

Whenever a DSA is allocated for the innermost procedure or procedures (at the same depth) of a nest of procedures, the optimization technique will try to meet the requirement from the reserved area. If this is not possible (because the DSA requires more than 512 bytes), the required storage is obtained in the standard way, using IHESADA.

A DSA allocated in the reserved area, or a DSA allocated in STATIC storage at compile time, is identified by a 'one' in the first bit of the second byte. (See IBM System/360 Operating System: PL/I (F) Compiler, Program Logic Manual for a discussion of DSAs in STATIC storage.)

CHAPTER 5: PL/I OBJECT PROGRAM MANAGEMENT (MULTITASKING)

This chapter describes the facilities provided by the PL/I Library for the dynamic management of PL/I multitasking programs in an operating system with MVT. A new task is created by the control program in response to an ATTACH macro instruction; the control program sets up a task control block (TCB), which contains all the control information related to the task; it may also set up an event control block (ECB), in which completion of the task will be posted. The new task then competes with other tasks for control according to the priority assigned to it. On completion of a task, the attaching task must remove the subtask's TCB from the system by issuing a DETACH macro instruction; if no ECB was set up and no end-of-task exit routine (ETXR) was specified, the DETACH macro instruction is unnecessary, and the TCB is removed from the system by the control program on termination of the task.

The tasks created in a PL/I multitasking program are executed as subtasks of a common ancestor, the control task. The use of a control task ensures that there is always present a task with a higher priority than that of the major task; the control task can then be entered whenever it is necessary to terminate the major task (e.g., on execution of a STOP statement). For multitasking , the program management module IHESA is replaced entirely by the module IHETSA; the user of a non-multitasking program incurs no significant overhead, since IHETSA is loaded only during link-editing of a multitasking program. Although some of the routines in IHETSA are peculiar to multitasking, most of them perform similar functions to the corresponding routines of IHESA; Figure 25 compares the two modules. Only those features of IHETSA that are not included in IHESA are described in detail. The library facilities for the multitasking pseudovariables and built-in functions, and for the WAIT statement, are described at the end of this section; Appendix K gives full

	Entry Points	
Function	IHESA	IHETSA
Get DSA	IHESADA	IHETSAD (Alias)
Get VDA	IHESADB	IHETSAV
Get controlled variable	IHESADD	See Note
Get LWS	IHESADE	IHETSAL
Get library VDA	IHESADF	IHETSAW
END	IHESAFA	IHETSAE
RETURN	IHESAFB	IHETSAR
GO TO	IHESAFC	IHETSAG
Free VDA/Free LWS	IHESAFD	IHETSAF
Free controlled variable	IHESAFF	See Note
Abnormal program termination	IHESAFQ	IHETSAY
Program initialization	IHESAPA IHESAPB IHESAPC IHESAPD	IHETSAP (Name) IHETSAA (Alias)
Environment modification	IHESARA	IHETSAN
Setting of return code	IHESARC	IHETSAC
Initialization of major task		IHETSAM
Initialization of subtask		IHETSAS
CALL with task option		IHETSAT
ETXR (end-of-task exit routine)		IHETSAX
Abnormal task termination		IHETSAZ

Note: The allocation and freeing of controlled storage in a multitasking environment is handled by a separate module, IHETCV, which is called by compiled code.

Figure 25. Comparison of IHESA and IHETSA

details of the PL/I control blocks for multitasking.

Control Task

The control task is entered via one of the initialization routines (IHETSAA and IHETSAP), and is established at a priority (16*JSPRI+11), where JSPRI is the priority specified in the JOB statement for the PL/I program. The entry point that is used depends on whether the PL/I program is called from an assembler-language routine. The control task obtains contiguous storage for its own save area and workspace, and for the PRV VDA for the major task. (If a PRV longer than 4096 bytes is requested, a message is printed out on the console and the program is terminated.) The length required for the PRV VDA is the sum of:

8 control bytes

L(PRV) (passed by the linkage editor)

L(LWS) (assembled by the initialization routine)

4 task-oriented control words

The format of the save area and workspace for the control task is shown in Figure 26.

Having allocated these storage areas, the control task:

1. Sets the STOP event control block to zero.

2. Creates a task variable for the major task, sets it active and initializes it, using an EXTRACT macro instruction to obtain the limit and dispatching priorities from the TCB set up by the operating system for the control task. (The task variable contains the task control information required by the PL/I Library.)

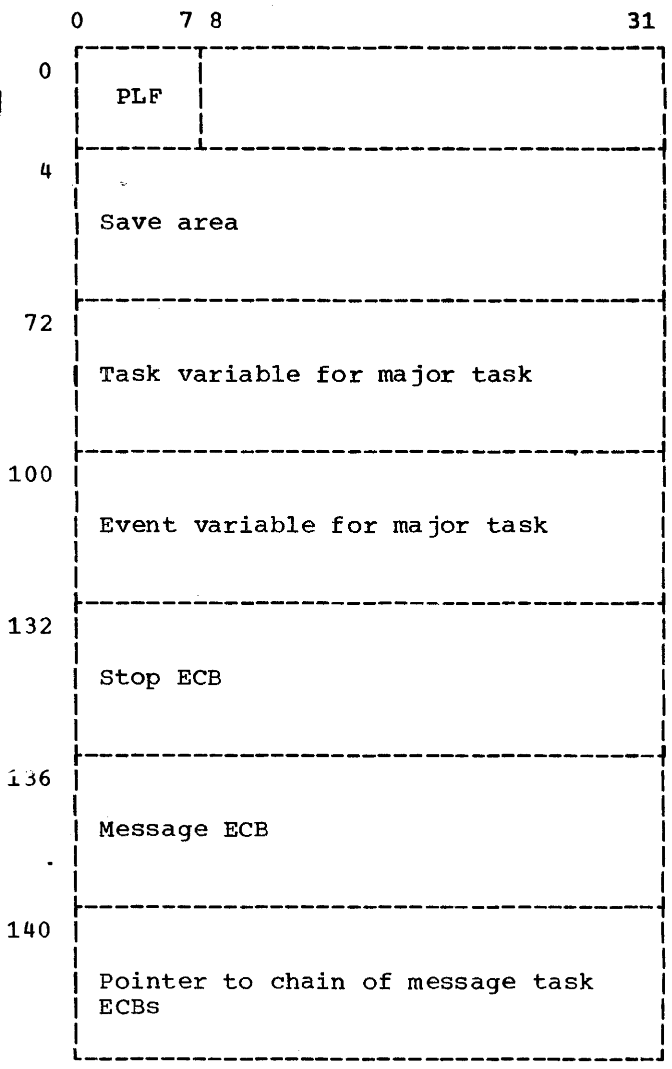
3. Creates an event variable for the major task, and sets it active.

4. Sets the ECB for the major task (which is contained in the event variable) to zero.

5. Sets the message ECB to zero. This will be posted by the ETXR routine (IHETSAX) in the event of a task terminating abnormally, so that the control task can attach a message task to put out a message.

6. Sets to zero the pointer to the chain of message task ECBs.

7. Sets the Program Lockout Flag (PLF) to zero (see Section on Multiprocessing at the end of this chapter).



0
7 8
31
0
PLF
4
Save area
72
Task variable for major task
100
Event variable for major task
132
Stop ECB
136
Message ECB
140
Pointer to chain of message task ECBs

•Figure 26. Format of Save Area and Workspace for Control Task

The control task next issues an IDENTIFY macro instruction to identify the major-task and subtask initialization routines, IHETSAM and IHETSAS, and the message task, so that these may later be attached. Finally it places in its save area the argument list that it will pass to IHETSAM, and sets the address of the save area in register RA.

To attach the major task, the control task issues an ATTACH macro instruction using IHETSAM as an entry point and giving the address of the ECB in the event variable of the major task. The control task shares subpool 1 with the major task so that, on completion of the major task, its

PRV VDA is still available. No end-of-task exit routine (ETXR) is specified, since control will return to the control task on termination of the major task. The action of the major-task initialization module IHETSAM is described under 'Initialization of Major Task'.

Having attached the major task, the control task issues a WAIT macro instruction which is to be satisfied when either

1. the STOP ECB is completed (i.e., when a STOP statement is executed), or
2. the ECB of the major task is completed (i.e., when the major task terminates normally or abnormally), or
3. the message ECB is completed (i.e., a message is to be displayed stating that a task has terminated abnormally).

If a task terminates abnormally, the ETXR routine (IHETSAX), posts the message ECB with a completion code equal to the address of an area of storage which it has obtained and which contains a save area and information for the message task. The control task then attaches a message task, sets the message ECB to zero, and returns to the WAIT macro as before. However, before the message task is attached, an area of storage is obtained to contain the ECB for the message task. This allows the message task to be waited on in the event of the major task terminating while the message task is still active. This area of storage is added to a chain which is pointed to by a word in the control task workspace.

The message task links to IHETEXB to put out the message, after which it frees the storage obtained for it by the ETXR routine.

The message is put out on SYSPRINT if it is open, otherwise it is put out on the console.

When the major task is completed normally, or when it is completed abnormally as a result of a PL/I error, the control task detaches the major task's TCB, frees subpool 1, and returns control to the calling program. The return code reflects the normal or abnormal termination of the program; if an operating-system interrupt has occurred, a message to this effect is printed out on the console, and the return code is the operating-system completion code.

If the major task has not been completed (i.e., if a STOP statement has been executed), the end-of-program routine IHETSAY terminates the major task and all its subtasks, and then posts the STOP ECB so that the control task gains control. The control task frees subpool 1 and then returns control to the calling program.

Initialization of Major Task

When the major-task initialization routine, IHETSAM, is attached, storage has already been allocated to the PRV VDA for the major task. IHETSAM is similar to the non-multitasking initialization routine IHESAP (described in Chapter 4), but in addition:

1. A flag bit (bit 8) in the PRV VDA is set to indicate that it is a multitasking PRV VDA.
2. The address of the task variable is placed in the PRV VDA, and the other task-oriented words of the PRV VDA are set to zero. (See Appendix K.)
3. After the standard action of initializing the PRV and LWS and setting the pseudo-registers IHEQVDA, IHEQFVD, and IHEQADC, the priority of the major task is reduced by one. This has the effect of making the whole program appear to have a priority one less than the operating-system limit priority (16*JSPRI+11), and enables the priority to be raised whenever it is essential that a routine be noninterruptible; it also allows the control task to be posted and entered immediately if necessary.

CALL with Task Options

When a CALL statement with a TASK, EVENT or PRIORITY option is executed, compiled code calls the library module IHETSAT to initialize the task and event variables for the subtask and to attach the subtask initialization routine IHETSAS. At compile time, if the TASK option had been specified, the compiler would have created a TASK variable, set it inactive, and inserted the addresses of the associated symbol table entry and event variable; if the EVENT option had been specified, the compiler would have created an event variable, set it inactive and set the STATUS halfword to zero. Futhermore, compiled code would have created an argument list (Figure 27) and inserted its address in register RA.

```
    0                                       31
   ┌─────────┬───────────────────────────────┐
  0│ Flags   │        A(Task variable)       │ Zero if no TASK option
   ├─────────┴───────────────────────────────┤
  4│             A(Event variable)           │ Zero if no EVENT option
   ├─────────────────────────────────────────┤
  8│ Priority relative to attaching task     │ Flags = X'80' if no PRIORITY option
   ├─────────────────────────────────────────┤
 12│             A(called procedure)         │
   ├─────────────────────────────────────────┤
 16│             For library use             │
   ├─────────────────────────────────────────┤
 20│             For library use             │ X'80' if no argument list
   ├─────────────────────────────────────────┤
 24│ Argument list for called procedure      │
   │ (X'80' in first byte of last entry      │
   │ indicates end of list)                  │
   └─────────────────────────────────────────┘
```
Figure 27. Parameter List for IHETSAT

IHETSAT raises the priority of the attaching task to the limit to ensure that the sequence cannot be interrupted by the current program, and then obtains a VDA, in which it places a remote parameter list for the execute form of the ATTACH macro instruction that it uses to attach IHETSAS. It then checks for the presence of the task and event variables; if either is present and active, the ERROR condition is raised. If either of the variables is absent (i.e., if the TASK or EVENT option were not specified), dummy task and event variables are placed in a VDA and initialized. Pointers to the PRV and DSA of the attaching task are stored in the two words of the parameter list reserved for library use; these are for reference by the subtask.

If the CALL statement includes a PRIORITY option, the sum of the relative priority from the parameter list supplied by compiled code and the dispatching priority in the task variable of the attaching task is stored in the task variable of the subtask; if the sum exceeds the limit priority for the PL/I program (16*JSPRI+10), the dispatching priority for the subtask is made equal to the limit. (See IBM System /360 Operating System: PL/I(F) Programmer's Guide for a discussion of priority of a PL/I program.) The limit priority of the attaching task is also placed in the task variable of the subtask. If there is no PRIORITY option, and a task variable exists, the dispatching priority in the task variable is assumed; if the task has a dummy task variable, the dispatching priority is the same as that of the attaching task at the time the subtask is attached.

To create the new subtask, IHETSAT issues an ATTACH macro instruction with the following parameters:

EP = SUB (the name given to entry point IHETSAS when it was identified).

ECB = A(ECB in subtask event variable)

ETXR = IHETSAX

No change in priority is made at this point. When control returns to IHETSAT, which is normally almost immediately, the address of the TCB for the new subtask, which is placed in register RA by the control program, is stored in the task variable for the subtask. IHETSAT then reduces the priority of the attaching task to its original level and returns control to the attaching task.

Initialization of Subtask

The subtask initialization routine IHETSAS is entered via an ATTACH macro instruction issued by IHETSAT; register RA contains the address of the parameter list prepared by compiled code (Figure 27). Since the priority of the subtask is at its limit, having been set there by IHETSAT, the subtask will gain control as soon as the priority of the attaching task is reduced at the end of the IHETSAT routine.

IHETSAS calculates the length of the PRV VDA required by the subtask, issues a GETMAIN macro instruction for the amount of storage needed (rounded up to a multiple of 2048 bytes), and then initializes the PRV VDA as follows:

1. It copies the contents of the PRV of the attaching task into the PRV of the subtask.

2. It copies any ON fields in the DSA of

the attaching task, and the procedure argument list (if one is being passed), into the PRV VDA of the subtask.

3. It increments the pseudo-register IHEQTIC by one. (IHETSAM sets IHEQTIC to zero when it initializes the major task. Each time a subtask is attached, IHETSAS adds one to the count in IHEQTIC; the count thus indicates the level of the task within the hierarchy.)

4. It initializes the new LWS and updates the pseudo-registers pointing at the various areas in LWS to their new values.

Having obtained storage and initialized the PRV VDA, IHETSAS executes the standard initialization routine as in a non-multitasking program, places the address of the procedure parameter list for the new subtask in register RA, reduces the priority of the subtask to the level given in its task variable, and branches to the address of the called procedure.

End-of-Task Exit Routine (IHETSAX)

When a subtask is attached, the end-of-task exit routine IHETSAX is specified in the ETXR operand of the ATTACH macro instruction. This routine is entered after the subtask has been completed; it is part of the attaching task, and is executed asynchronously with it. If the subtask was terminated by the PL/I storage-management routines, the only function of IHETSAX is to detach the TCB of the subtask.

If the subtask was completed abnormally by the operating system, an area of storage is obtained in which the name of the subtask and the completion code are stored. This storage area also contains space for a save area to be used by the message task. IHETSAX then posts (using the POST macro) the message ECB in the control task storage area. The control task receives control and attaches a task which prints a message giving the name (if any) of the subtask, the operating system completion code, and, in the more common cases, an indication of the probable error. When IHETSAX regains control, it detaches the TCB of the subtask.

To obtain the name of the subtask for insertion in the message, IHETSAX locates the subtask's task variable by initiating a save-area trace from the current task's external save area, the address of which is in the current task's TCB. It obtains the completion code from the subtask's TCB.

GO TO Statements

The multitasking housekeeping routine for GO TO statements (IHETSAG) differs from its non-multitasking equivalent only in that if control is transferred outside the block in which the statement occurs, any tasks attached in the blocks that are freed must be terminated. If any tasks have been attached in the block, the task variable chain pointer in the DSA will point to the task variable of the most recently created subtask. IHETSAG searches the chain through each DSA in each task until a task is found that has attached no subtasks; this task is then terminated. The process is repeated until all tasks attached in the block, and their descendants, have been terminated. In the process, all storage associated with these tasks is returned to the supervisor, and all files opened in the tasks are closed.

On-Units and Entry Parameter Procedures

The multitasking routine IHETSAN modifies a recursive environment when an on-unit or an entry parameter procedure is entered or ended. It differs from the non-multitasking routine (IHESARA) in two respects:

1. The chain of recursive DSAs is followed back to the PRV of the major task.

2. If a CALL statement calls an entry parameter procedure with a task option, the address of the entry parameter is placed at the top of the parameter list, the address of IHETSAT is assigned to the entry parameter, and IHETSAN is called. When IHETSAN terminates, it points register RA at the IHETSAT parameter list and branches to IHETSAT.

Termination of a Task

A PL/I task can be terminated by the execution of any one of the statements END, RETURN, STOP, and EXIT.

The action taken by the library END (IHETSAE) and RETURN (IHETSAR) routines is similar to that of the GO TO routine (IHETSAG); the action differs from that of the non-multitasking equivalents in that any tasks attached in the block being terminated must also be terminated. If the block to be terminated is also the end of a procedure called with a task option, subpool 1 (automatic and controlled storage) is freed and control is returned to the control program. If it is the end of the major task, the FINISH condition is raised and the program branches to the error-handling routine. When the END or RETURN routine has been completed, control is returned to the control program, but subpool 1 is not released. (Automatic storage is required by the control task; controlled storage may be required by the calling program.)

The abnormal-end-of-task routine (IHETSAZ) is entered

1. from IHEERR when return is made from the ERROR routine in a subtask or from the FINISH routine in the major task,

2. when an EXIT statement is executed in any task, or

3. when CALL IHEDUMT is executed in any task.

IHETSAZ detaches the task, and any tasks that it has attached, in the manner described under 'GO TO Statements', places a return code in the task's ECB, and returns control to the control program.

The end-of-program routine (IHETSAY) is entered when a STOP or CALL IHEDUMP statement is executed in any task. IHETSAY terminates all subtasks in the manner described under 'GO TO Statements', and then passes control to the control task by posting the STOP ECB; the control task then terminates the major task.

The completion code in the STOP ECB or the ECB for the major task indicates how the program was terminated.

Controlled Storage

The allocation and freeing of storage for controlled variables in a multitasking environment is handled by library module IHETCV. This module is independent of IHETSA and is loaded only if the CONTROLLED attribute is used. When storage is allocated, the task invocation count from pseudo-register IHEQTIC is stored in the first halfword of the controlled variable. Before a controlled variable is freed, its task invocation count is checked; if it does not correspond with the value in IHEQTIC for the task in which the statement occurs, the variable is not freed. Controlled storage is allocated in subpool 0 if it is in the major task, and in subpool 1 if it is in a subtask.

MULTITASKING PSEUDO-VARIABLES AND BUILT-IN FUNCTIONS

Statements in which the STATUS pseudo-variable appears, or which contain the COMPLETION or STATUS built-in functions, are executed from compiled code without a library call.

COMPLETION Pseudo-Variable

On execution of an assignment statement in which the COMPLETION pseudo-variable appears, the expression on the right-hand side is evaluated and converted to a bit string of length 1, which is then stored at bit 24 of a fullword. Compiled code then calls IHETEVA, passing the address of the event variable named in the pseudo-variable, and that of the fullword (in a list pointed to by register RA). If the event variable is active, the ERROR condition is raised; otherwise IHETEVA takes the following action:

1. It raises the priority of the current task to the limit to prevent interruption.

2. It sets the I/O flag in the event variable (bit 1 of the flag byte) to zero.

3. If the bit string = '0'B, it sets bit 1 (the 'complete' bit) of the ECB in the event variable to zero, restores the priority of the task to its original level, and returns control to the task.

4. If the bit string = '1' B, it tests to see whether the event is already complete. If it is, IHETEVA restores the priority of the task to its original level and returns control to the task; otherwise it posts the ECB with a completion code of zero, restores the priority, and returns control to the task.

PRIORITY Pseudo-Variable

The PRIORITY pseudo-variable is used to set the dispatching priority of a task to a new value relative to that of the current task. On execution of an assignment statement in which the PRIORITY pseudo-variable appears, the expression on the right-hand side is evaluated and converted to a fixed-point binary constant of default precision, which is assigned to a fullword. Compiled code then calls IHETPRA, passing the address of the task variable of the task named in the pseudo-variable and that of the fullword (in a list pointed to by register RA). If the pseudo-variable does not specify a task, the current task is assumed. IHETPRA raises the priority of the current task to the limit to prevent interruption, and accesses the dispatching priority from the task variable; it assigns to the task variable the new value of dispatching priority, calculated as follows:

```
    New dispatching priority of named task
      =MAX (0,MIN(limit-1,P+N))

    where P=dispatching priority of current
            task
      and N=increment
```

If the task whose priority is being changed is not the current task, IHETPRA restores the priority of the current task and returns control to it.

If the priority of the current task is changed, after the new priority has been stored in the task variable a CHAP macro instruction is issued to change the priority of the task before returning control.

PRIORITY Built-In Function

The PRIORITY built-in function yields the dispatching priority of a task relative to that of the current task. On execution of a statement in which the function appears, compiled code calls IHETPBA, passing the address of the task variable of the task named in the function and the address of a fullword target field (in a list pointed to by register RA). IHETPBA subtracts the dispatching priority of the current task from that of the named task, and assigns the difference to the target field. The dispatching priorities are obtained from the respective task variables.

THE WAIT STATEMENT

When a WAIT statement is executed in a multitasking environment, compiled code calls the library module IHETSW, passing the addresses of the event variables associated with the statement. IHETSW scans the event variables to see whether enough events to satisfy the WAIT statement are PL/I complete ('complete' bit, ECMP, set to 1). If not, IHETSW scans the ECBs for the I/O events, and in each case where the I/O event is complete sets the check bit (EMCH) in the corresponding event variable to 1; a list is then made of all the incomplete I/O and multitasking events.

If the number of PL/I and I/O complete events is sufficient to satisfy the WAIT statement, the relevant I/O transmit modules are invoked to complete the I/O events. (See 'General Logic and Flow' under 'Record-Oriented I/O' in Chapter 3.) If there are no multitasking events in the list, and if the number of completed I/O events is not sufficient and all the I/O events must be completed to satisfy the WAIT statement, the check bit in each event variable is set to 1 and the relevant I/O transmit module is invoked. If not all the I/O events need to be waited on, or if there are some multitasking events in the list, a multiple WAIT macro instruction is issued for the list of incomplete events. When the macro has been satisfied, if the list included any I/O events, the corresponding ECBs are scanned and the check bits in the event variables corresponding to completed ECBs set to 1; the I/O transmit module is then invoked.

The I/O event variables that are checked by the transmit modules are set complete and the check bits are set to zero. The event variables are then set inactive and removed from the task and file chains.

In a non-multitasking environment, library module IHEOSW is called by complied code. This module is similar to IHETSW except that it only accepts I/O event variables and inactive event variables.

Alternative I/O Modules for Multitasking Programs

Alternative multitasking and non-multitasking modules for input/output operations have been created in order to prevent the non-multitasking user from being inflicted with any multitasking overheads. These modules are:

Non-multitasking	Multitasking
IHEOCL	IHEOCT
IHECLT	IHECTT
IHEPRT	IHEPTT
IHEIOB	IHEIBT
IHEDDO	IHEDDT
IHEION	IHEINT

The entry points for the multitasking modules correspond with the entry points of the non-multitasking modules. Modules which have no alternative form will call the correct module by extracting its address from the list addressed by pseudo-register IHEQADC. This list is assembled into IHESA or IHETSA, whichever is present.

MULTIPROCESSING

Since raising the priority of a task to the limit priority on a multiprocessing machine does not ensure that no other task is executing simultaneously, additional precautions must be taken when performing certain operations to prevent two tasks accessing the same control blocks simultaneously.

These operations are: manipulation of EVENT variables; termination of tasks while still active; task attachment; updating chains associated with EXCLUSIVE files; and changing the priority of a subtask.

The following control blocks are used, in conjunction with raising the priority to the limit to prevent simultaneous access.

Program Lockout Flag (PLF): This is a one byte flag located in the first byte of the control task's storage, and is known to all tasks. It is set to zero at program initialization time.

Must Complete Flag (MCF), Wait to Terminate Flag (WTF): These are one byte flags associated with a particular task and are located in that task's EVENT variable.

Wait to Terminate ECB (WTE), Infinite Wait ECB (IWE): There are fullwords also associated with a particular task and are located in that task's EVENT variable.

Exclusive File Flag (EFF): This is a one byte flag associated with a particular EXCLUSIVE file and is located in the file's FCB.

EVENT variables

The PLF is used during operations involving EVENT variables. Before any operation involving an EVENT variable is begun, the priority of the current task is raised to the limit. Since no I/O or macro instructions are executed until the operation is finished, this task will not lose control until after it has restored its priority to the original value. A TS instruction is issued on the PLF, and if the latter is already set, the task loops on the TS instruction until it is turned off. Hence if a task, which is executing simultaneously, is also performing an operation on an EVENT variable, the first task will loop until the second task has completed its operation. On completion of the EVENT variable operation, the PLF is set to zero and the priority of the task restored to its original value.

Must Complete Operations

A Must Complete Operation is an operation which, once begun by a task, must be allowed to complete before that task can be terminated by a higher level task. These are: task attachment; normal task termination; and all operations involving the PLF or an EFF.

Before beginning a Must Complete Operation, a task first tests its WTF. If it is on then the task is about to be terminated by a higher level task (see Task Termination below) and so it waits on its IWE until terminated. If its WTF is zero, the task sets its MCF on, raises its priority to the limit and proceeds with its Must Complete Operation. When the operation is complete, the task tests its WTE to see if a task is waiting for it to complete its Must Complete Operation. If a task is waiting, the task which has completed its Must Complete Operation POSTs the WTE and waits on its IWE until terminated. If no task is waiting it resets its MCF to zero, restores its priority and continues.

Task Termination

If a task A is terminating an active subtask B, it first of all sets B's WTF. It then tests B's MCF. If it is on, then B is not in a position to be terminated (i.e., it is doing a Must Complete Operation) and so A issues a WAIT on B's WTE. When B completes its Must Complete Operation, it tests its WTE to see if a task is waiting and if so it POSTs the WTE and waits on its IWE until terminated. When A comes out of the wait state due to B's POST, it can then go ahead and terminate B.

If A had found that B was not executing a Must Complete Operation, then it would go straight ahead and terminate B. Should B then wish to start a Must Complete Operation, it would first test its WTF, find it on, and then wait on its IWE until terminated.

EXCLUSIVE Files

The EFF is used in a similar manner to the PLF. When a task wishes to update chains associated with a particular EXCLUSIVE file, it issues a TS instruction on the EFF associated with that file. Any other task wishing to do a similar operation with the same file will then loop until the first task has reset the EFF to zero.

Task Attachment

The initialization of a subtask involves accessing the attachor's storage, and to ensure that the subtask has completed its initialization before the storage is changed, the attachor WAITs on the subtask's IWE immediately after attaching it. The subtask POSTs the ECB when it has completed the initialization.

Changing Priorities

In order to prevent the priority pseudo-variable routine from changing the priority of a task which is at limit priority, the routine first tests the TCB of the subtask whose priority it is changing to see if it is at limit priority. If so, it must wait until the subtask has restored its original priority. Hence it waits on the subtask's IWE and when the subtask has restored its priority it tests its IWE to see if the priority routine is waiting. If not, it POSTs the IWE and the priority routine can then go ahead and change the subtask's priority.

The PL/I Library handles two types of conditions at object time which cause interruption to the main flow of a program. These are:

1. Conditions for which it is possible to specify an on-unit:

   a. Computational program interrupts.
   b. Other conditions.

2. Execution error conditions not covered by a PL/I-defined condition.

If any of these conditions occurs, control is passed to the library error handling module IHEERR. This module is always resident; if it is necessary to print a message at execution time, IHEERR links to a group of modules normally non-resident but brought into storage when required. These are:

IHEESM:	This loads one of the message modules and prints the appropriate message.
IHEERD:	Data processing error messages.
IHEERE:	Error messages other than those in the other error message modules.
IHEERI:	Input/output error messages.
IHEERO:	Error messages for non-I/O ON conditions.
IHEERP:	Error messages for I/O ON conditions.
IHEERT:	Multitasking error messages.

The error messages and their associated ONCODES are described in IBM System/360 Operating System: PL/I (F) Programmer's Guide.

All the PL/I-specified ON conditions except I/O SIZE and I/O CONVERSION are raised by compiled code to facilitate reference by the error-handling subroutines. Each ON condition has a code number (internal to the library) consisting of two hexadecimal digits. When an ON condition is raised, the code associated with it is placed in the error-handling pseudo-register IHEQERR.

There is an error message for each ON condition. In some cases the condition (e.g., CONVERSION) may have a group of errors associated with it and has therefore a group of messages. A complete list of the internal error codes and their associated messages is given in Appendix E.

PROGRAM INTERRUPTS

Fifteen possible program interrupts can occur in System/360. Seven of these are, or may be, related to computational conditions in PL/I (see Figure 28); on-units may be specified for these conditions. Seven of the remaining eight are treated as errors of a non-ON type; significance is not handled.

Program Interrupts	PL/I Conditions
Fixed-point overflow	FIXEDOVERFLOW
Fixed-point divide	ZERODIVIDE
Decimal overflow	FIXEDOVERFLOW
Decimal divide	ZERODIVIDE
Exponent overflow	OVERFLOW
Exponent underflow	UNDERFLOW
Floating-point divide	ZERODIVIDE

Figure 28. Program Interrupts and PL/I Conditions

Because the user may specify on-units for handling certain PL/I conditions, when an interrupt occurs the PL/I program must gain control to see if there is an on-unit associated with that particular interrupt. This is achieved by the Get PRV subroutine in the IHESA module, which issues a SPIE macro to:

1. Provide a program interrupt control area (PICA). This is a six-byte area (in IHESA) which contains the address to which control is passed when an interrupt occurs, and information on the type of interrupt handled by IHEERR.

2. Cause the supervisor to create a program interrupt element (PIE). This is a 32-byte area which contains the PICA address and also a save area for the old PSW and registers 14 to 2 when an interrupt occurs.

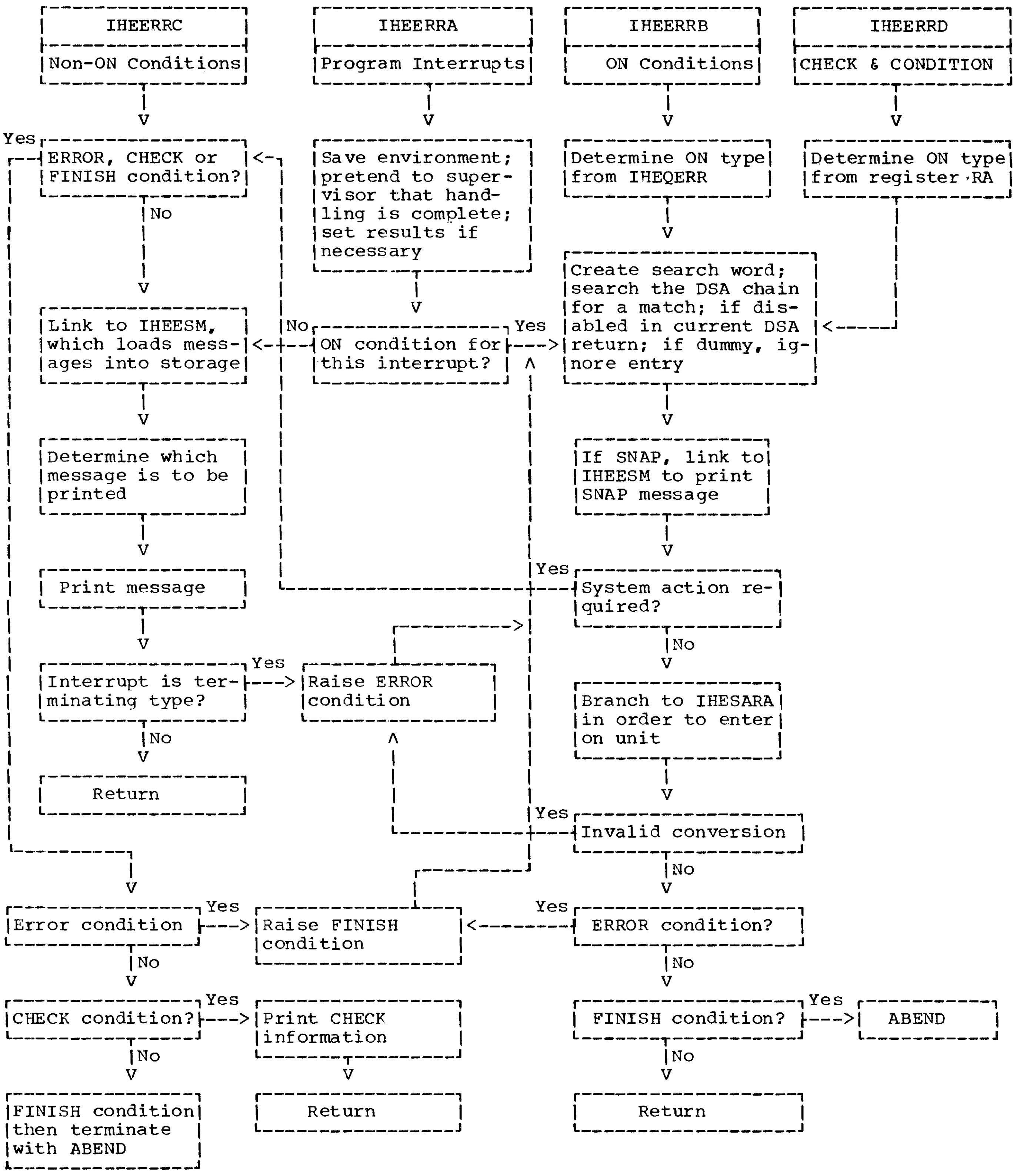

IHEERRC
Non-ON Conditions
IHEERRA
Program Interrupts
IHEERRB
ON Conditions
IHEERRD
CHECK & CONDITION
Yes
ERROR, CHECK or FINISH condition?
No
Save environment; pretend to supervisor that handling is complete; set results if necessary
Determine ON type from IHEQERR
Determine ON type from register RA
Create search word; search the DSA chain for a match; if disabled in current DSA return; if dummy, ignore entry
Link to IHEESM, which loads messages into storage
No
ON condition for this interrupt?
Yes
If SNAP, link to IHEESM to print SNAP message
Determine which message is to be printed
Print message
Yes
System action required?
No
Yes
Interrupt is terminating type?
Raise ERROR condition
No
Branch to IHESARA in order to enter on unit
Return
Yes
Invalid conversion
No
Yes
Error condition
Raise FINISH condition
Yes
ERROR condition?
No
No
Yes
CHECK condition?
Print CHECK information
Yes
FINISH condition?
ABEND
No
No
FINISH condition then terminate with ABEND
Return
Return

Figure 29. Flow through the Error Handling routine (IHEERR)

```
 0 3 4   7 8                           31
┌───┬────┬──────────────────────────────┐
│   │ PM │     A(Exit subroutine)       │
└───┴────┴──────────────────────────────┘

 32                  47
┌─────────────────────┐
│   Interrupt Mask    │
└─────────────────────┘
```

Figure 30. Format of the Program Interrupt Control Area (PICA)

<u>Definitions of PICA fields:</u>

PM: Program mask

A(Exit subroutine): Address of the entry point in IHEERR to which control is to be passed when one of the specified interrupts occurs. This entry point is IHEERRA.

Interrupt mask: Indicates to the supervisor which interrupts are to be handled by IHEERR. These interrupts are all the fifteen possible ones except significance.

```
 0       7 8                               31
┌─────────┬─────────────────────────────────┐
│         │             A(PICA)             │
├─────────┴─────────────────────────────────┤
│             OPSW(Bits 0-31)               │
├───────────────────────────────────────────┤
│             OPSW(Bits 32-63)              │
├───────────────────────────────────────────┤
│               Register 14                 │
├───────────────────────────────────────────┤
│               Register 15                 │
├───────────────────────────────────────────┤
│               Register 0                  │
├───────────────────────────────────────────┤
│               Register 1                  │
├───────────────────────────────────────────┤
│               Register 2                  │
└───────────────────────────────────────────┘
```

Figure 31. Format of the Program Interrupt Element (PIE)

<u>Definitions of PIE fields:</u>

A(PICA): Address of PICA, for supervisor use

OPSW: Contents of the old program status word

Registers 14 to 2: Contents of these registers when an interrupt occurs

On entry to IHEERRA, register RA contains the address of PIE.

It is possible for another program interrupt to occur before user corrective action has been completed. IHEERR has to guard against this eventuality when it obtains control, otherwise the second interrupt would cause the supervisor to terminate the task. To avoid this, the following method is used:

1. The PSW in PIE (the old PSW) is saved in the LWE area in library workspace.

2. Bits 40 to 63 of the PSW in PIE are changed to contain the address of the appropriate entry point in IHEERR; control is returned to the supervisor.

3. The supervisor assumes the interrupt has been handled satisfactorily and transfers control to the new address in the PSW in PIE; thus it enters module IHEERR again.

Floating-point registers are saved in the library communication area, and the old PSW is inspected to find the cause of the interrupt.

If a fixed-point or decimal overflow interrupt is forced to occur, the SIZE condition may be raised. Therefore when one of these interrupts occurs, the pseudo-register IHEQERR must be inspected to see if the SIZE code has been set. Similarly, if any of the divide interrupts occurs, IHEQERR must be inspected to see if the ZERODIVIDE code has been set. If it has, the condition is disabled and control returns to the point of interrupt.

Certain very unusual circumstances may result in a program interrupt occurring during the execution of IHEERR or of one of the library modules called, or linked to, from it. For example, if the program destroys the PRV, or the DSA chain, or parts of library workspace, then it is likely that sooner or later a specification or addressing interrupt will occur.

Under these circumstances, the programmer or systems engineer requires a dump at the earliest opportunity. To achieve this, and to prevent any attempt to re-enter IHEERRA on account of the second interrupt, a SPIE macro is issued every time IHEERR is entered. This macro provides that, in the event of an interrupt occurring, IHEERR shall be entered at entry point IHEERRE. Similarly, another SPIE macro is issued at each exit point, to restore IHEERRA as the normal entry point for program interrupts during the execution of compiled code and library routines.

When IHEERRE is entered, a message is printed on the console and the program is abnormally terminated, with a dump.

Type	Condition	Condition Prefixes permitted	Default situation
Computational	CONVERSION FIXEDOVERFLOW OVERFLOW SIZE UNDERFLOW ZERODIVIDE	Yes	All enabled except SIZE
List processing	AREA	No	Always enabled
Input/ Output	ENDFILE ENDPAGE KEY NAME RECORD TRANSMIT UNDEFINEDFILE	No	Always enabled
Program checkout	CHECK SUBSCRIPT-RANGE STRINGRANGE	Yes	Disabled
Programmer-named	CONDITION	No	Always enabled
System action	ERROR FINISH	No	Always enabled

Figure 32. PL/I ON Conditions

ON CONDITIONS

The six classes of ON conditions defined in PL/I are shown in Figure 32. To deal satisfactorily with the situation when any of these conditions arise, IHEERR must:

1. Recognize the condition.
2. See if it is enabled.
3. If so, see if there is an on-unit for the condition.
4. If there is an on-unit, transfer control to IHESARA, which, after doing the necessary housekeeping, will transfer control to the on-unit.
5. If no on-unit, take system action for the condition.
6. Return to the interrupted program or terminate, according to the provisions of the PL/I language.

In order to carry out these operations IHEERR needs:

1. Information passed when the error condition arises.
2. Information set by compiled code in the DSA for each procedure. A two-word ON field is allocated in the DSA for this purpose. (See Chapter 4.)

Action by Compiled Code

Action taken by compiled code in preparation for the possibility of a condition arising during execution is summarized here.

Prologue: The prologue allocates space in the DSA for:

1. Every ON statement in the block.
2. Each ON condition disabled in the block.

ON CHECK (identifier 1,......identifier n) is interpreted as n ON statements.

For each of the occurrences given above, the prologue stores information in the two words in the DSA ON field:

1st word: Contains the error code for the condition and the address of data identifying the condition. This word is called the search word comparator. (See Figure 33.)

Type of ON condition	Contents of word	
	Byte 1	Bytes 2 to 4
I/O	Error code	A (DCLCB)
CONDITION	Error code	A (CSECT)
CHECK (label) CHECK (variable)	Error code	A (Symbol name & length) A (Symbol table)
Others	Error code	Nothing stored

Figure 33. Format of the Search Word comparator

2nd word: Bits 0, 1 and 4 of the first byte are set as follows:

Bit 0 = 0 Not the last ON field in the DSA

= 1 Last ON field in the DSA

Bit 1 = 1 Condition disabled

Bit 4 = 1 Dummy ON field

In the second word, either bit 1 or bit 4 is set to 1. (See 'Prefix Options', below.)

ON Statement: When the ON statement is executed, compiled code stores information in the second word of the ON field:

Byte 1:

```
Bit 2 = 0 SNAP not required
      = 1 SNAP required

Bit 3 = 0 Normal
      = 1 System action required

Bit 4 = 0 No longer dummy
```

Bytes 2-4: A(on-unit)

Prefix options: An ON field for an ON condition must be created by the prologue whenever:

1. An ON statement is present in the block.

2. An ON condition becomes disabled at any time during the execution of the block.

3. CHECK is enabled within the block.

This ON field is always set to dummy by the prologue. It is also set to disabled if:

1. The condition is disabled by a prefix option in the block-header statement.

2. The condition is disabled by default and there is no enabling prefix option in the block-header statement, or within the block. The exceptions to this are CHECK, SIZE, STRINGRANGE, and SUBSCRIPTRANGE, which are dealt with as follows:

   CHECK: No ON fields are created if this condition is disabled by default

   SIZE, STRINGRANGE, and SUBSCRIPTRANGE: If these conditions are disabled by default, flags are set in the flag byte of the DSA as follows:

SIZE:	bit 7 = 0
STRINGRANGE	bit 2 = 0
SUBSCRIPTRANGE:	bit 4 = 0

Execution of an ON statement in the block causes removal of the dummy flag and insertion of the flags indicating the action required. It does not remove the disable flag if on. Execution of a REVERT statement causes reinstatement of the dummy flag.

During execution of the block, statements may be executed which have disabling prefix options in them. Compiled code must be inserted before and after the statements to:

1. Set the disable flag before the statement.

2. Restore the original flags after the statement.

Similarly, to enable prefix options, compiled code must:

1. Set the disable flag off before the statement.

2. Restore the original flags after the statement.

Prefix options specified on outer blocks carry down into internal blocks. The implementation of these blocks should be as if the option had been explicit in each of them.

Action by the Library

When an ON condition arises during execution, IHEERR gains control from one of the following:

1. The supervisor

2. Compiled code

3. Another library module

In case 1, the ON condition code required is determined by inspection of the program interrupt code in the old PSW. For cases 2 and 3, the ON condition code is passed in pseudo-register IHEQERR, except for the CHECK and CONDITION conditions, when a parameter list is used. From this code and information passed in the calling sequence, a search word is generated in library workspace in all three cases; the format of the search word is identical with that of the search word comparator (Figure 33).

When the search word has been created, IHEERR initiates a search through the chain of DSAs to determine the action to be taken. Each DSA is analyzed in turn, from the end of the chain upwards towards the beginning. The search proceeds as follows:

1. Bit 6 of the flag byte of the first available DSA is tested to see if that DSA contains any ON fields. Then:

   a. No ON fields: If the DSA is the current DSA and the condition is SIZE, STRINGRANGE, or SUBSCRIPTRANGE, the flag byte of this DSA is examined to see if the condition is disabled:

      Disabled: the program returns to the point of interrupt.

      Not disabled: The DSA is ignored.

      If the condition is CHECK, the program returns to the point of interrupt.

   b. ON fields: The first word of each ON field - the search word comparator - is compared with the search word to see if a match is found. If a match is found, the ON field in the DSA is tested to see what action is required.

2. If the last ON field is reached before finding a match, then:

   a. If the DSA is the current DSA and the condition is SIZE, STRINGRANGE, or SUBSCRIPTRANGE, the corresponding flags in the DSA are tested.

   b. The error code is tested to see if the condition is CHECK.

This may result in a return to the point of interrupt. If not, the next DSA is obtained and analyzed in the same way.

If a match has been found, then the following tests are made:

1. Is the condition disabled by a prefix option? (This test can only be applied when the matching ON field is contained in the current DSA.)

   Disabled: No further processing in IHEERR; the program returns to the point of interrupt.

   Not disabled: Next test is made.

2. Is the matching ON field a dummy ON field?

   Dummy ON field: The field is ignored and the next DSA is obtained.

   No dummy ON field: Next test is made.

3. Is SNAP action required?

   SNAP action required: A summary flow trace is written on the system output file. This output contains the ON-condition abbreviation and trace-back information identifying the procedures in the chain. The statement number may optionally be included. Each procedure is identified by chaining back through the DSA chain until a procedure DSA is found and then using the contents of register BR in the appropriate save area. The search ends when the chain-back reaches the external save area. An example of this output is given in IBM System/360 Operating System: PL/I (F) Programmer's Guide.

   SNAP action not required: Proceed normally.

In a multitasking program, when the search word has been created, IHEERR calls IHETER, which searches the ON fields of the DSA in a similar manner to IHEERR. In the absence of a matching ON field, the search continues until the PRV VDA of the major task is reached. If a subtask PRV VDA is encountered during the search, any ON fields that have been copied into it from the DSA of the attaching task are also checked. If a match is not found, the search continues through the DSAs of the attaching task.

System Action

System action means writing a message and then either continuing or raising the ERROR condition. It is performed if:

1. the system action flag is set in the matching ON field, or

2. no matching ON field can be found in the DSA chain.

If a match is found, and an on-unit address is given, then, to guard against the possibility of recursive use when control returns from the on-unit by means of a GO TO statement, a new block of library workspace is obtained. This LWS is added to the DSA chain as described in 'PL/I Object Program Management'. In order to pass control to the on-unit, the recursion subroutine in IHESA is called; this establishes the correct environment and then branches to the on-unit. Return from the on-unit may be made in one of two ways:

1. On normal completion, control passes

to IHEERR, which returns to compiled code at the point following the instruction which caused the condition to be raised.

2. Execution of a GO TO statement. In this case the GO TO subroutine (IHESAFC or IHETSAG) is entered to carry out the housekeeping described in Chapters 4 and 5.

STANDARD SYSTEM ACTION AND CONDITIONS OTHER THAN ON CONDITIONS

If an ON condition is raised and there is no matching ON field for the condition, standard system action is taken. This action is defined by the PL/I language. Another set of error conditions can arise at object time for which no specific ON condition is defined in the language (e.g., logarithm of a negative number). In these cases, implementation-defined system action is taken.

An error message is printed when PL/I-defined or implementation-defined system action occurs. Then, depending on the severity of the condition, either processing continues or the ERROR condition is raised. In a non-multitasking program, or in a major task, raising the ERROR condition generally leads to the FINISH condition being raised and then to the abnormal termination of the job step by the ABEND macro. The exceptions to this are when there is a GO TO statement in the ERROR or FINISH unit. In a multitasking program, if the ERROR condition is raised in a subtask, instead of the FINISH condition being raised, IHETSAZ is invoked. (See 'Termination of a Task' in Chapter 5.) A complete list of object-time error messages, with details of the conditions that cause them to be issued, is given in _IBM System/360 Operating System: PL/I (F) Programmer's Guide_.

When the printing of an error message is required, the appropriate modules of the non-resident part of the error package are dynamically loaded into storage. The seven modules concerned are:

IHEERD, IHEERE, IHEERI, IHEERO, IHEERP, IHEERT: The error message modules; they contain the error message texts together with tables to locate the messages. Only the module containing the required message is loaded.

IHEESM: Contains the code required to print SNAP and system action messages. This module is always required.

An action indicator is obtained during the process to determine whether normal processing should continue if the ERROR condition is raised. The appropriate action is taken when the message has been printed as output.

BUILT-IN FUNCTIONS

The two built-in functions, ONLOC and ONCODE, may only be used in an on-unit; they provide environmental information associated with the raising of the latest ON condition.

ONLOC

An interrupt can occur that can cause entry to the on-unit in which ONLOC is specified. If this happens, the ONLOC built-in function identifies the BCD name of the entry point of the procedure in which the interrupt occurs.

The address of this BCD name is computed by chaining back through the DSA chain until the first procedure DSA is reached and by using the contents of BR in the appropriate save area. The length of this name and the maximum length are found; these two lengths and the pointer to the BCD name are inserted in the target SDV whose address has been passed to ONLOC as a parameter.

If ONLOC is specified outside an on-unit, a null string is inserted in the target SDV.

ONCODE

The ONCODE built-in function picks up a value from the WONC field in the library communication area in LWS previously set by IHEERR. This value is implementation-defined by the type of error that caused the interruption. It may be specified in any on-unit. If specified in an ERROR or FINISH unit, the ONCODE will be that of the error or condition that caused the ERROR or FINISH unit to be entered.

If ONCODE is specified outside an on-unit, a unique ONCODE value (0) is returned. A list of ONCODEs and an explanation of their use are given in _IBM System/360 Operating System: PL/I(F) Programmer's Guide_.

MODEL 91 INTERRUPT HANDLING

Program interrupts occurring in code executed on an IBM System/360 Model 91 require different treatment from that described above. This is necessary because the Model 91 is capable of executing several instructions concurrently: hence a situation may arise in which several program exceptions may occur before an interrupt is raised.

As soon as a single exception occurs, the Model 91 ensures that execution of the instructions already decoded is completed, and then raises an interrupt. During execution of these instructions, further exceptions may occur. If there are no more instructions to be executed at the time an exception occurred, then the interrupt raised is known as a precise interrupt; the PSW contains the address of the instruction following that in which the exception occurred.

If, however, further instructions were executed, then the interrupt is known as an imprecise interrupt; the PSW at interrupt-time contains the address of the next instruction to be executed, but this is not necessarily the address of the instruction following any of the exceptions raised. The instructions causing the exceptions cannot therefore be identified. If there is more than one exception prior to interrupt, then a multiple-exception imprecise interrupt is said to have occurred. Full details of Model 91 operation and interrupt handling are given in IBM System/360 Model 91, Functional Characteristics, Form A22-6907.

When an imprecise interrupt is raised, therefore, the Model 91 indicates the situation by setting the interruption code and the interruption length code in the PSW as follows:

1. Recognition that an imprecise interrupt has occurred: Bits 26-33 are set to zero.

2. Identification of the type or types of exception in the interrupt: Bits 16-25 are set as follows:

Bit	Type of Exception
16	Protection
17	Addressing
18	Specification
19	Data
20	Fixed-point overflow
21	Fixed-point divide
22	Exponent overflow
23	Exponent underflow
24	Significance
25	Floating-point divide

Implementation

The Library module IHEM91 handles the problems associated with imprecise interrupts on a Model 91. This module is obtained by the user specifying the Model 91 option in his program; this creates an ESD entry that results in IHEM91 being linkage-edited with the Library error and interrupt module IHEERR.

Initially, IHEERR tests bits 26-31 of the PSW to determine if these bits are all zero (i.e., if an imprecise interrupt exists):

1. All zero: Imprecise interrupt; control is passed to IHEM91

2. Any bit non-zero: No imprecise interrupt; IHEERR handles the situation in the normal way

On receiving control, IHEM91 tests bits 16-25 to determine which exceptions have occurred. All bits (except significance) are tested, as more than one type of exception can occur in an imprecise interrupt. If the bit tested is on (non-zero), then:

1. Condition list: IHEM91 sets an entry in a list of PL/I conditions and program exceptions. The list is stored in the LWE area of Library workspace (LWS); an entry indicates that the particular condition or exception must be raised. The list consists of from one to eight entries, processed in the order:

   UNDERFLOW
   FIXEDOVERFLOW or SIZE
   OVERFLOW
   ZERODIVIDE
   Data exception
   Specification exception
   Addressing exception
   Protection exception

   Note: ZERODIVIDE is entered only once in the list, even if floating-point divide and fixed-point divide both occur. Significance is not handled, as it is disabled in all PL/I programs. FIXEDOVERFLOW and SIZE cannot both be raised since they are raised by the same hardware condition.

2. Interrupt count: The value in the ONCOUNT field (WONC + 4) in the LCA is incremented by 1. Thus the total value in this field is the total number of conditions or exceptions to be raised. When a multiple-exception imprecise interrupt does not exist (because there are no exceptions or only a single exception) the value in the ONCOUNT field is zero.

IHEM91 then returns control to IHEERR in order that each condition in the list can be raised. As described above, a condition can be handled in one of two ways:

1. By entering an ON-unit, with exit by either:

   a. A normal return
   b. A GO TO statement

2. By system action

These rules have to be considerably extended for handling a multiple-exception imprecise interrupt:

1. ON unit for UNDERFLOW, FIXEDOVERFLOW, SIZE, OVERFLOW or ZERODIVIDE:

   a. Normal return: Next entry in the list is processed. If there are no more entries to be processed, then a return is made to the address in the PSW.

   b. GO TO statement: No more entries in the list are processed, and no information indicating the nature of these unprocessed entries is given. However, the ONCOUNT built-in function, when used in an ON unit, will return the number of entries remaining unprocessed.

2. System action:

   a. For UNDERFLOW: When the error message has been printed, the next entry in the list is processed.

   b. For FIXEDOVERFLOW, SIZE, OVERFLOW, or ZERODIVIDE: No further entries in the list are processed. If the program terminates as an immediate result of system action, messages are printed to indicate the nature of the unprocessed entries.

3. ERROR raised for a data, specification, addressing or protection exception: No further entries in the list are processed. If the program terminates as an immediate result of the system action, messages are printed to indicate the nature of the unprocessed entries.

In order to implement these rules, IHEERR tests for a multiple-exception imprecise interrupt after:

1. Return from an ON unit: If a multiple-exception imprecise interrupt exists, IHEM91 is entered at a second entry point in order to:

   a. Process the next entry

   b. Reduce the ONCOUNT value by one

   c. Return to IHEERR

2. Program termination caused by ERROR condition: If a multiple-exception imprecise interrupt exits, IHEM91 is entered at a third entry point. The condition list is processed in order to print out a message for each entry not handled at the time the program terminated. Program termination is completed when the list is exhausted.

ONCOUNT Built-in Function

The ONCOUNT built-in function returns a non-zero value only when this function is used in an ON unit entered as a result of a multiple-exception imprecise interrupt in a Model 91. In such a situation, the binary integer returned is the number of entries that remain unprocessed (including the current one) at the time the ONCOUNT function is used.

Flush Instructions

A program may not operate correctly on the Model 91 if it requires identification of the instruction causing an imprecise interrupt. Similarly, it may not operate correctly if it requires that an imprecise interrupt is honored before some instruction later in the program is executed. However, the unwanted effects of imprecise interrupts can usually be eliminated by placing 'flush' instructions at certain points in the program. A 'flush' instruction is an Assembler Language instruction of the form:

```
BCR x,0
```

where x is not equal to zero. An instruction of this type is a no-operation instruction for all of System/360, but it is implemented in the Model 91 in such a way that its execution is delayed until all previously decoded instructions have been executed.

If the M91 compiler option is specified, flush instructions are generated by the compiler at the following points in the program:

1. Before every ON statement
2. Before every REVERT statement
3. Before code to set the SIZE condition
4. For every null statement
5. Before code to change prefix options.

If both the M91 and the STMT options are specified, the compiler generates a flush instruction to precede every statement in the program.

Model 91 Object-Time Diagnostic Messages

If object-time diagnostic messages are issued as a result of an imprecise interrupt, the words "AT OFFSET..." are replaced by "NEAR OFFSET...", since in these circumstances the instruction causing the interrupt cannot be precisely identified.

After a multiple-exception imprecise interrupt on a Model 91, certain exceptions will remain unprocessed if the ERROR condition is raised before all the exceptions have been handled. If the program subsequently terminates as a direct result of the ERROR condition being raised in these circumstances, one or more of the following messages will be printed out.

IHE810I	PROTECTION EXCEPTION UNPROCESSED AFTER MULTIPLE-EXCEPTION IMPRECISE INTERRUPT
IHE811I	ADDRESSING EXCEPTION UNPROCESSED AFTER MULTIPLE-EXCEPTION IMPRECISE INTERRUPT
IHE812I	SPECIFICATION EXCEPTION UNPROCESSED AFTER MULTIPLE-EXCEPTION IMPRECISE INTERRUPT
IHE813I	DATA EXCEPTION UNPROCESSED AFTER MULTIPLE-EXCEPTION IMPRECISE INTERRUPT
IHE814I	ZERODIVIDE UNPROCESSED AFTER MULTIPLE-EXCEPTION IMPRECISE INTERRUPT
IHE815I	OVERFLOW UNPROCESSED AFTER MULTIPLE-EXCEPTION IMPRECISE INTERRUPT

One function of the PL/I Library is to provide a standard interface with the control program which can be utilized by compiled code. Detailed implementation is described in Chapters 3, 4, and 5. The implementation described here concerns support for PL/I language statements and functions with a control program interface that does not fall into one of the categories discussed in those chapters. These are the PL/I statements DISPLAY, DELAY, STOP and EXIT, and the built-in functions TIME and DATE.

Full and Minimum Control Systems

The full control system of IBM System/360 Operating System will enable the PL/I Library to issue macro instructions which support the above-mentioned statements and functions. The relationship is as follows:

PL/I	Macro instruction
DELAY	STIMER(WAIT)
TIME	TIME
DATE	TIME
DISPLAY	WTO, WTOR (WAIT)

Thus, the library support for language features is as follows:

DELAY: The execution of the current task is suspended for the required time.

EXIT and STOP: Both these statements raise the FINISH condition and then cause termination of the PL/I program.

TIME: The time of day is returned to the caller in the form HHMMSStht where:

HH =	hours (24-hour clock)
MM =	minutes
SS =	seconds
tht =	tenths, hundredths and thousandths of a second

DATE: The date is returned to the caller in the form YYMMDD where:

YY =	year
MM =	month
DD =	day

DISPLAY: A message may be written on the console with no interruption in execution or, if a reply is expected, execution is suspended until the operator's reply is received. If the EVENT option is used when a reply is expected, execution is continued without interruption until a corresponding WAIT statement is encountered; execution is then suspended until a reply is received.

The minimum control system does not support the TIME and STIMER macro instructions. Use of the DELAY statement, and TIME and DATE built-in functions will result in the ERROR conditions being raised.

I/O EDITING AND DATA CONVERSION

PL/I allows the user a wide choice in selecting the representation for his data, both on the external medium and internally in storage; considerable flexibility is permitted in specifying changes of data type and form. The library conversion package is designed to implement the full set of editing and conversion functions. To avoid unnecessary duplication of code, standard intermediate forms are used. This has the effect of reducing the number of library modules in the package to about fifty, to cover about two hundred logical conversions. To speed up processing, direct routines are provided for some of the most frequently used conversions, while the compiler generates in-line code for some of the simpler ones.

To restrict further the storage requirements for the library conversion package, the F level compiler analyses the actual changes of data required for a particular execution. Sometimes these are not fully known at compile time, and then a worst case has to be taken. From this information, by use of the linkage editor LIBRARY statement and external references within the compiled modules, the loading of conversion modules is limited to those known to be required. This technique can be of considerable value, especially when only a small number of data types is used by the source programmer. Further details are provided in IBM System/360 Operating System: PL/I (F) Compiler, Program Logic Manual.

With one exception, all the modules contained within the library conversion package are called by means of the PL/I standard calling sequence (described in 'Linkage Conventions', Chapter 2). The exception is IHEVCS (complex-to-string director) which is called by the operating system external standard calling sequence.

The letters in the module name indicate the module usage; see Figure 34.

STRUCTURE OF LIBRARY CONVERSION PACKAGE

To perform a change from a source data item to a target data item may involve a succession of steps and the use of several individual library modules within the package. The structure of the library conversion package is shown in Figure 36.

In association with each individual step, the attributes of the source or the target fields, or of both, must be known. The required information is provided in the calling sequences. Each data item has a corresponding format element descriptor (FED) or data element descriptor (DED). With one exception, the formats of these control blocks are described in Appendix H. The exception is that of a DED generated at object time for communication between library modules. (See Figure 35.)

Letters						Meaning
1	2	3	4	5	6	
I	H	E	D			Director
I	H	E	K			Picture check
I	H	E	V	P		Conversion involving packed-decimal intermediate, except IHEVPG and IHEVPH
I	H	E	V	F		Conversion involving floating-point intermediate
I	H	E	V	K		Conversion involving numeric fields
I	H	E	V	S		Conversion involving strings
I	H	E	V	C		Conversion involving external character data being converted to type string
I	H	E	V	Q		Direct conversion to improve performance
I	H	E	U	P		Mode conversions

Figure 34. Module Usage indicated by Letters of Module Name

Code	Bit 0	1	2	3	4	5	6	7
= 0	1	1	Non-sterling	Short	1	Decimal	Fixed	Real
= 1	1	1	Sterling	Long	1	Binary	Float	Complex

Note: Bits 0, 1 and 4 are always 1.

Figure 35. DED Flag Byte for Character Representation of an Arithmetic Data Item

This DED is created when it is necessary to convert a character representation of an arithmetic value to an intermediate coded arithmetic data type, prior to conversion to a string target. The form of this DED is the same as that for a coded arithmetic data item (CAD), and consists of a flag byte and precision bytes representing the quantities p and q. As for coded data, the flag byte defines the attributes of the corresponding data item; bit 1 is set to 1 to indicate that a character representation of an arithmetic value is referred to.

Directors

The structure chart makes frequent reference to 'directors'. These modules are used to fulfil two main purposes:

1. The matching of source element with target element, which may not be known at compile time.

2. The controlling of the flow at object time by means of interpretative information passed to them.

The latter function is best illustrated by the arithmetic conversion director (IHEDMA), where a single call determines the flow through a sub-package of over twenty arithmetic conversion routines. (See below in 'Arithmetic Conversions'.)

There are director routines at four levels. (See Figure 36.) They are:

1. Complex format directors.

2. Input/output format directors and the complex-to-string director.

3. String-to-arithmetic and arithmetic-to-string directors.

4. Arithmetic conversion director.

All directors except the complex-to-string directc can be called directly from compiled code; the complex-to-string director is invoked from the complex format directors or from list/data-directed input only.

Any director can call any below it in the structure.

Edit-directed I/O

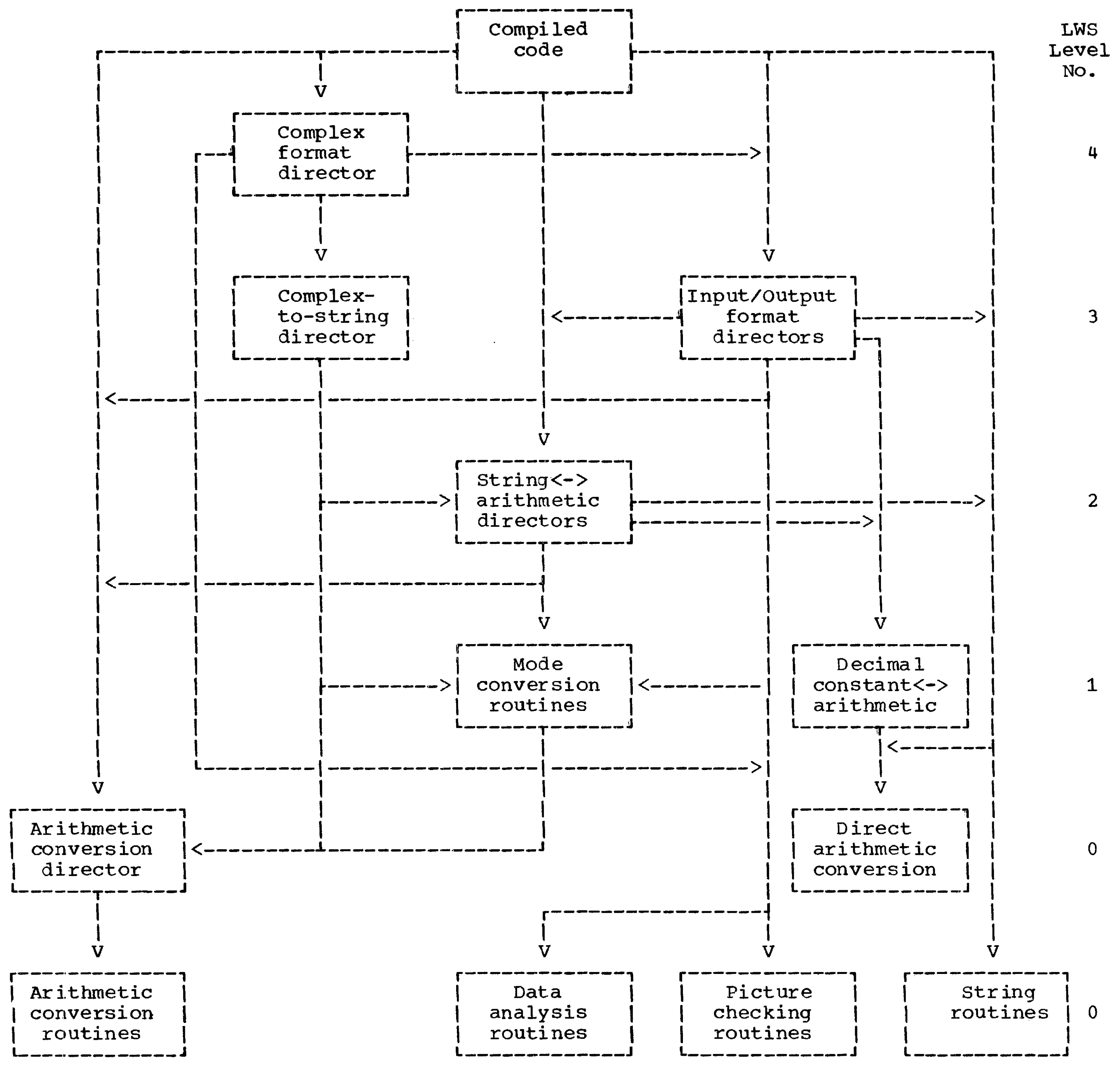
Edit-directed transmission allows the user to specify the storage area to which data is to be assigned or from which data is to be transmitted and the actual form of the data on the external medium. The information concerning storage areas is specified in the source program by means of a data list, and the information about the form of the data on the external medium by means of a format list.

The library conversion package is designed to implement the executable format scheme discussed in Chapter 3. This is done by the object time matching of list item and format item through the use of the director routines mentioned above. The set of I/O directors provided and their association with the PL/I data format items is shown in Figure 37.

I/O EDITING

Complex Directors: Complex format items on the external medium may have real and imaginary parts of differing attributes. When the list item and the target field are of type arithmetic, this situation is handled in the complex director by making consecutive calls for real and imaginary format items, and passing control to the particular format director associated with the format item.

When the target field is a string, however, there are two problems with C format items. First, the data on the



Compiled code
LWS Level No.
Complex format director
4
Complex-to-string director
Input/Output format directors
3
String<-> arithmetic directors
2
Mode conversion routines
Decimal constant<-> arithmetic
1
Arithmetic conversion director
Direct arithmetic conversion
0
Arithmetic conversion routines
Data analysis routines
Picture checking routines
String routines
0

Note: <-> indicates a conversion in either direction

Figure 36. Structure of the Conversion Package

external medium must be scanned dynamically in order to deduce the attributes of the format item. The information derived from this is stored in a special DED. (See 'Structure of Library Conversion Package'.) This DED is necessary for the conversion of all format items and constants.

Second, the base, scale and precision of the real and imaginary parts have to be compared, to determine the highest set of attributes, so that the form of the converted data in the string target may be known. This is done by invoking a special director, called the complex-to-string director, which performs the necessary analysis on the DEDs of the real and imaginary parts of the C format item. Each item is then converted by the rules of type conversion to coded complex and then to string.

Input/Output Directors: The input/output directors named above (other than C format) perform three major functions. Because there are slight differences between input and output, the functions are described under these headings.

Input: A call is made to IHEIOD to request w bytes and a data field pointer. If the w bytes can be obtained from the current buffer, the address returned to the input director is that of the data field in the buffer itself. If not, a VDA is obtained and the requisite field of w bytes is built up in the dynamic area. The VDA address is stored in WSDV in the LCA.

These two conditions are normal. If, on the other hand, an abnormal return occurs at this point, this signifies that an ENDFILE condition exists and that a return has been made from an ENDFILE on-unit. In this case, the I/O director must return control to the code associated with the next PL/I source statement, which is point-

PL/I format item	Director	Module name Input	Module name Output
Complex	C	IHEDIM	IHEDOM
Fixed and floating point	F/E	IHEDIA	IHEDOA
Bit string	B	IHEDID	IHEDOD
Character string	A	IHEDIB	IHEDOB

Picture	P(DEC,STL)
	P(CHAR)
		IHEDIE
		IHEDIB
			IHEDOE
			IHEDOB

Figure 37. Input/Output Directors for PL/I Format Items

INPUT		
String value	List item	Conversion
Character string	Arithmetic Character string Bit string	Character to arithmetic Character string assignment Character to bit string
Bit string	Arithmetic Character string Bit string	Bit string to arithmetic Bit string to character string Bit string assignment
Arithmetic (including expression)	Arithmetic Character string Bit string	Arithmetic type conversion Arithmetic to character string Arithmetic to bit string
OUTPUT		
List item	String value	Conversion
Arithmetic	Character representation of data value	Arithmetic to character string
Bit string	Bit string in character form	Bit to character
Character string	Character string	Character string assignment

Figure 38. Conversion for List/Data Directed I/O

ed at by the second word of pseudo-register IHEQCFL.

If there is no abnormal return, the target DED is inspected by the director routine and the first stage of the necessary conversion process is initiated by means of a suitable call to a routine below the input director level. (See structure chart, Figure 36.)

When the conversion has been completed and the data item assigned to the list item, the input director calls the I/O package again. At this stage, the I/O routine tests for the TRANSMIT condition, and, if necessary, calls IHEERR, to specify that the TRANSMIT condition is active, and that the format item transmitted is therefore suspect. In addition, any VDA that has been allocated is freed.

Output: A call is made to the library I/O package to obtain an address for the external data item. If the w bytes specified can be satisfied within the current buffer, the address of the current buffer pointer is returned; if not, a VDA is obtained and the address of this dynamic storage is passed back. The source DED is then inspected and a call is made to the first subroutine in the conversion package to perform conversion.

After assignment of the data item to a buffer area or VDA, a call to the appropriate I/O routine is made from the output director. If a VDA was used, the output field is split off into the appropriate buffers and the dynamic storage released.

For both input and output, control is finally returned to compiled code.

List- and Data-directed Input/Output

The total set of conversions required by list/data-directed I/O is shown in Figure 38.

Since all the conversions represented deal with change of data from one internal representation to another, the conversion package is fully capable of performing the conversion for list/data-directed I/O. The type conversions are fully defined in the PL/I language and the modules that implement them are given below. Some examples of list/data-directed I/O are included in IBM System/360 Operating System: PL/I (F) Programmer's Guide.

MODE CONVERSIONS

Since data may be declared COMPLEX, and complex values may be written or read by list-directed and data-directed input and output, or by the C format item, two routines are provided to facilitate conversions of mode during I/O editing and during conversions between internal arithmetic and string data.

TYPE CONVERSIONS

Four director routines are provided to control the flow which enables changes between data of type string and data of type arithmetic, as required by the PL/I language. These routines are used by list-, edit- and data-directed I/O and in some internal conversions.

	TO:		
	Arithmetic	String	
		Bit	Character
FROM:			
Arithmetic	-	IHEDNB	IHEDNC
Bit string	IHEDBN	-	-
Character string	IHEDCN	-	-

Figure 39. Modules for Type Conversions

STRING CONVERSIONS

A set of generalized interpretive routines is provided to support the possible string conversions and assignments that may exist. Each module interrogates source and target information contained in the string dope vectors and DEDs in order to handle truncation, padding, and alignment for fixed and varying strings. Figure 39 shows the modules provided; it should be noted that there is no difference between a source character string with a picture and one without, as once the data has been checked into the source field, no further use is made of the picture.

	TO:		
	Bit	Character	Character with picture
FROM:			
Bit	IHEVSA	IHEVSB	IHEVSF
Character	IHEVSD	IHEVSC	IHEVSE

Figure 40. Modules for String Conversions

ARITHMETIC CONVERSIONS

A direct routine IHEVQA converts floating-point data to fixed-point binary, in order to provide fast processing of this frequently used routine. Normally, however, all conversions (including this one) are dealt with by the library conversion package.

This package carries out editing and conversions for all type arithmetic source fields which have type arithmetic target fields. It also handles conversions of format items and constants, which are character representations of arithmetic type data. The flow control through this subpackage is achieved by the arithmetic conversion director described below.

The method employed is to use an intermediate form of representation according to the form of the source data and to relate this intermediate form to the target data, either by direct conversion or by use of a second intermediate form (which implies radix change). The two intermediate forms in use are:

1. Packed decimal intermediate (PDI)

   This consists of 17 digits and a sign, together with a one-word scale factor (WSCF) in binary representing powers of ten.

2. Long floating-point intermediate (FPI)

   This is the standard internal form, and consists of 14 hexadecimal digits.

The logical flow through the package is shown in Figure 41.

The arithmetic conversion director (IHEDMA) links together the modules required for a particular arithmetic conversion. It is called either directly by compiled code or by other director routines. The flag bytes in the source and target DEDs are interrogated to determine which modules are required for the current conversion and their order of execution. The library communication area is used to record information required by successive modules as follows:

WBR1	Address of entry point of second module
WBR2	Address of entry point of third module (if required)
WRCD	Target information

The conversion director then passes control to the first module in the chain; the first transfers control to the second, and so on until the conversion is complete. The last module returns to the program which called the conversion director. All the modules which can be first in the chain set up by the conversion director use the source parameters passed to this director. The first conversion is always to the intermediate form of the same radix as the source. The results are stored in the following LCA fields:

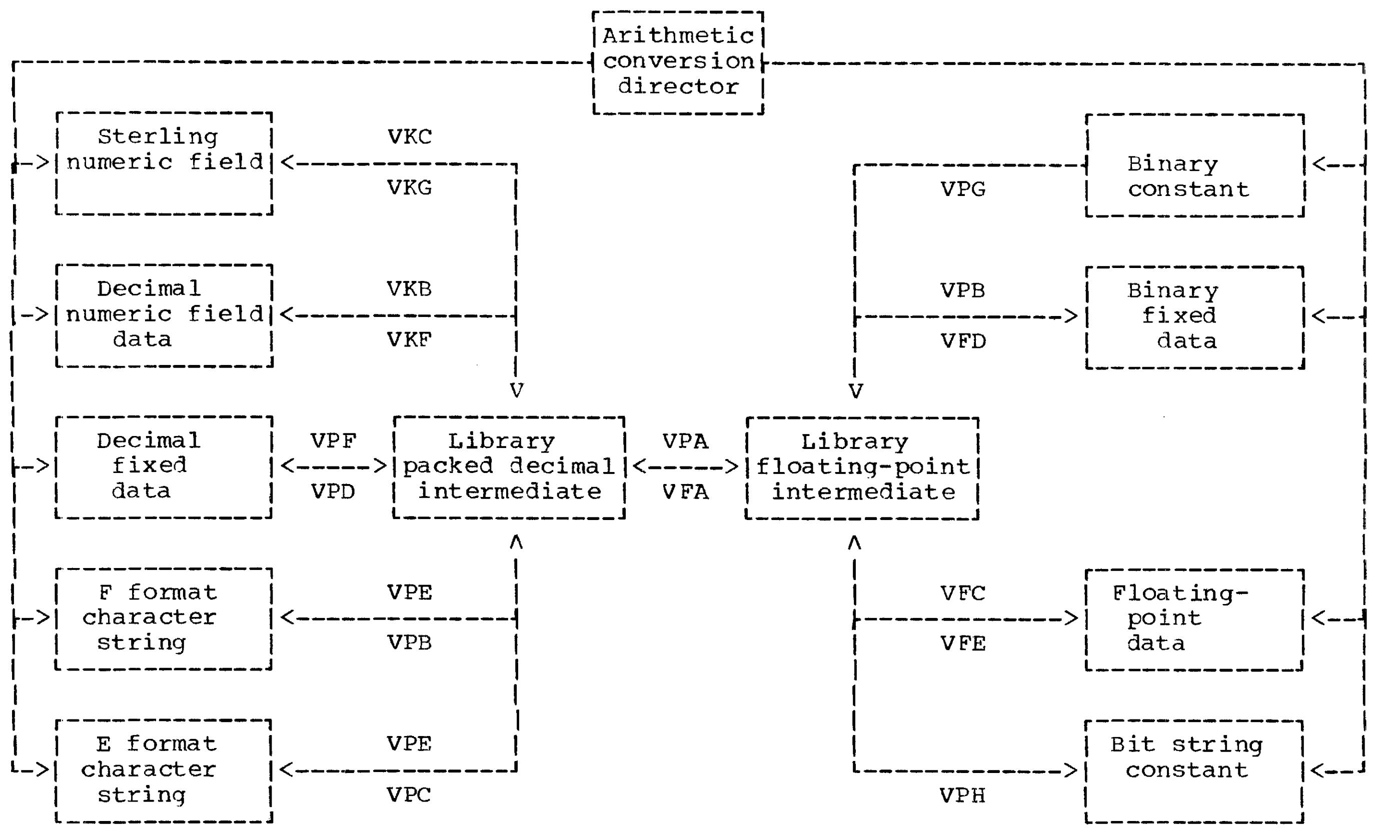
WINT	Binary results
WINT WSCF	Decimal results

Three modules in the arithmetic package deal with data on the external medium. Two modules handle the output of F and E format items from packed decimal intermediate format, and the third provides conversion from F or E format items to packed decimal intermediate format. The LCA fields used for these modules are:

WFED	A(FED) at input
WFDT	A(FED) at output
WSWA WSWC	Switches
WOCH	A(Error character): for ONCHAR built-in function
WOFD	Dope vector for ONSOURCE built-in function

DATA CHECKING AND ERROR HANDLING

Checking is carried out on data on the external medium for edit-, data- and list-directed input and on internal data items taking part in conversions.


Arithmetic conversion director
Sterling numeric field
VKC
VKG
Decimal numeric field data
VKB
VKF
Decimal fixed data
VPF
VPD
Library packed decimal intermediate
VPA
VFA
Library floating-point intermediate
F format character string
VPE
VPB
E format character string
VPE
VPC
Binary constant
VPG
Binary fixed data
VPB
VFD
Floating-point data
VFC
VFE
Bit string constant
VPH

Note: The three-letter names, e.g., VKC, are the last three letters of the module name. A name above the flow lines indicates a conversion from left to right; a name below the line indicates a conversion from right to left.

Figure 41. Structure of the Arithmeric Conversion Package

Edit Directed

All data described by a picture is matched against the picture description. When a P format item is read in, this checking is performed by one of three picture check routines (decimal, sterling, and character) which is called by the appropriate input director.

F/E format items are checked against the format element descriptor (FED). The validity of the characters in the data item is investigated prior to conversion to packed decimal intermediate format.

If B format items are assigned in the target DED to a bit string, the items are checked in the character-to-bit module. Otherwise, a pre-scan within the B format input director checks that all characters in the string are either zero or one.

If A format or B format is specified on input without a w specification, the compiled code calls IHEDIL (illegal-input format director). This routine calls the execution error package, passing an error code. This causes a message to be printed and the ERROR condition to be raised.

List/Data-Directed

Within the conversion package, the constants which are converted to arithmetic are checked in the appropriate internal conversion modules.

Decimal constants are converted by the F/E-to-PDI routine and are therefore checked by that routine as above.

Binary constants are checked prior to conversion to floating-point intermediate.

Bit string constants are checked prior to conversion to floating-point intermediate.

Internal Conversions

Checking of data is provided for the following:

1. Character string to arithmetic.

2. Character string to bit string.

3. Character string to pictured character string.

4. Bit string to pictured character string.

In cases 1 to 3 above, if an invalid character is found the CONVERSION condition is raised; in case 4, the ERROR condition is raised.

When CONVERSION is raised, an error code is passed to IHEERR. The error code passed depends:

1. On the type of operation (internal, I/O, or I/O with TRANSMIT condition raised).

2. On the various formats and conversions involved. These consist of:

   F format
   E format
   B format
   Character string to arithmetic
   Character string to bit string
   Character string to pictured character string
   P format (decimal, character and sterling)

Different ONCODE values are set for each, and may be interrogated in an on-unit provided for the CONVERSION condition. If the condition is associated with I/O, it is also possible that a TRANSMIT condition may be active. This can be tested in the on-unit for CONVERSION. A list of ONCODE values is given in IBM System/360 Operating System: PL/I (F) Programmer's Guide.

The conversion package routines set the following information before invoking the execution error package:

WOFD	Dope vector for field scanned
WOCH	Address of character in error
IHEQERR	Value of the error code. For I/O editing, a 1 bit is set in bit zero. Bits 12 to 15 are set according to the conversion being performed. (See Figure 42.)

Conversion	Code
F format	1
E format	2
B format	3
Character string to arithmetic	4
Character string to bit string	5
Character string to pictured character string	6
P format (decimal)	7
P format (character)	8
P format (sterling)	9

Figure 42. Conversion Code Set in IHEQERR

In addition to the occurrence of the CONVERSION error, the SIZE condition can also occur in the conversion package. Once again, a distinction is made between internal conversions and conversions involving the external medium. In the latter case, bit zero in IHEQERR is again set to one.

In certain cases an illegal conversion may be requested or an invalid parameter may be passed to a conversion routine. In these cases the conversion package calls the error-handling subroutine, having set register RA to point to an error code. This causes a message to be printed which describes the error found; the error-handling subroutine then raises the ERROR condition.

If a CONVERSION error occurs, the program can proceed in three ways:

1. If system action is specified, a message will be printed and the ERROR condition raised.

2. If CONVERSION is disabled, the conversion will continue, ignoring the character in error.

3. If an on-unit exists, it will be entered. If the on-unit returns control to the conversion routines, they will assume that either the ONCHAR or ONSOURCE pseudo-variable has been used to correct or replace the character or field in error, and will automatically retry the conversion.

Note: If the pseudo-variables have not been used to correct the error, and if the on-unit attempts to return control to the conversion, a message will be printed and the ERROR condition raised.

COMPUTATIONAL SUBROUTINES

Computational subroutines within the PL/I Library supplement compiled code in the implementation of operators and functions within four main groups. These groups are:

1. String handling
2. Arithmetic evaluation
3. Mathematical functions
4. Array functions

In addition to the description provided in this document, detailed information on algorithms and performance is published in IBM System/360 Operating System: PL/I Subroutine Library: Computational Subroutines.

A number of error and exceptional conditions not directly covered by PL/I-defined ON conditions may occur in these subroutines. In these cases, a diagnostic message is printed and the ERROR condition raised. By use of the ONCODE built-in function, the cause of interrupt may be ascertained in an ERROR unit and appropriate action may be taken. A list of the error messages and ONCODEs is given in IBM System/360 Operating System: PL/I (F) Programmer's Guide.

When an aggregate of data items is being processed, the indexing through the aggregate is achieved by in-line code, as the library routines generally handle individual elements only. The array functions, however, perform their own indexing, so that only a single call from compiled code is made.

For modules handling data in coded form, character six of the module name indicates the type of data concerned; the meanings of this character are given in Figure 43.

Data	Character		
Internal form	Real	Complex	Real or Complex
Binary	B	U	
Packed decimal	D	V	
Binary or packed decimal	F	X	
Short float	S	W	G
Long float	L	Z	H

Figure 43. Relationship of Data Form and Sixth Character of Module Name

STRING OPERATIONS AND FUNCTIONS

The library string package contains modules for handling both bit and character strings. Generally, individual modules handle a particular function or operation for bit or for character string; in the interests of efficiency however, additional modules are provided to deal with byte-aligned data for some of the bit string operations.

The functions LENGTH and UNSPEC are handled directly by compiled code; support for BIT and CHAR is provided in the library conversion package.

Linkage to the string subroutines is by means of the operating system standard for the functions SUBSTR, INDEX and BOOL, and by the PL/I standard for all others. The functions REPEAT, HIGH, and LOW use the PL/I standard as they are implemented as entry points to the concatenation and assign/fill routines.

The address and the maximum and current lengths of a string are passed to library modules by means of string dope vectors. All string lengths supplied in SDVs are assumed to be valid non-negative values; unpredictable results will ensue if this condition is not satisfied.

Conversions (e.g. of decimal integers into binary integers for functions such as REPEAT) and evaluation of expressions are handled by the compiler, which is also responsible for recognising instances of byte-alignment which are suitable for the byte-aligned bit functions provided.

The general design of the string package is influenced by the concept that complete evaluation of the right-hand side of an assignment statement occurs before the assignment. In this evaluation, there is usually an intermediate stage in which a partial result is placed in a field acting as a temporary result field. This does not prevent the compiler from optimizing by providing the actual target field of the assignment as the temporary result field, subject to the following conditions:

1. If the target field is the same as a field involved in expression evaluation, an intermediate area is required to develop the result (unless otherwise stated in the module description summaries). For example, A = B || A requires an intermediate field, but A = A & B does not.

PL/I Operation	PL/I Function	Bit String		Character String
		General	Byte-aligned	
And	-	Use BOOL	IHEBSA	-
Or	-	Use BOOL	IHEBSO	-
Not	-	Use BOOL	IHEBSN	-
Concatenate	REPEAT	IHEBSK	-	IHECSK
Compare	-	IHEBSD	IHEBSC	IHECSC
Assign	-	IHEBSK	IHEBSM	IHECSM
Fill	-	IHEBSM	-	IHECSM
-	HIGH/LOW	-	-	IHECSM
-	SUBSTR	IHEBSS	-	IHECSS
-	INDEX	IHEBSI	-	IHECSI
-	BOOL	IHEBSF	-	-

Figure 44. String Operations and Functions

ARITHMETIC OPERATIONS				
Operation	Binary fixed	Decimal fixed	Short float	Long float
Real Operations				
Integer exponentiation: x**n	IHEXIB	IHEXID	IHEXIS	IHEXIL
General exponentiation: x**y	-	-	IHEXXS	IHEXXL
Shift-and-assign, Shift-and-load	-	IHEAPD	-	-
Complex Operations				
Multiplication/division: z_1*z_2, z_1/z_2	IHEMZU	IHEMZV	-	-
Multiplication: z_1*z_2	-	-	IHEMZW	IHEMZZ
Division: z_1/z_2	-	-	IHEDZW	IHEDZZ
Integer exponentiation: z**n	IHEXIU	IHEXIV	IHEXIW	IHEXIZ
General exponentiation: z_1**z_2	-	-	IHEXXW	IHEXXZ

ARITHMETIC FUNCTIONS				
Function	Binary fixed	Decimal fixed	Short float	Long float
Real Arguments				
MAX, MIN	IHEMXB	IHEMXD	IHEMXS	IHEMXL
ADD	-	IHEADD	-	-
Complex Arguments				
ADD	-	IHEADV	-	-
MULTIPLY	IHEMPU	IHEMPV	-	-
DIVIDE	IHEDVU	IHEDVV	-	-
ABS	IHEABU	IHEABV	IHEABW	IHEABZ

Figure 45. Arithmetic Operations and Functions

2. Padding of fixed-length strings does not occur automatically when a string operation is performed, except in the case of assignment of fixed-length character strings and fixed-length byte-aligned bit strings. Separate routines are available for padding.

ARITHMETIC OPERATIONS AND FUNCTIONS

Library arithmetic modules provide support for all those arithmetic generic functions and operations for which the F level compiler neither generates in-line code nor (as for the functions FIXED, FLOAT, BINARY, and DECIMAL) uses the library conversion package.

Linkage between compiled code and the arithmetic modules is established by means of the operating system standard for the functions supported and by means of the PL/I standard for the operators supported. The module description summaries provide information about linkage to individual modules.

Fixed-point data often require data element descriptors (DEDs) to be passed in order to convey information about precision (p, q). Binary data is always assumed to be stored in a fullword correctly aligned, with $0 < p \leq 31$. Decimal data is always assumed to be packed in FLOOR (p/2) + 1 bytes, where $0 < p \leq 15$. Where such fields introduce high-order digits beyond the specified precision, these digits must not be significant.

In decimal routines, the target area is assumed to be of the correct size to accommodate the result precision as defined by the language.

Where assignment to a smaller field is required, the compiled code should generate an intermediate field for the result and subsequently make the assignment. This does not apply to ADD, MULTIPLY and DIVIDE with fixed-point decimal arguments, which perform the assignment themselves. Such action by compiled code avoids much unnecessary object-time testing and enables a clear distinction to be made between SIZE and FIXEDOVERFLOW conditions.

Floating-point arguments are assumed to be normalized in aligned fullword or doubleword fields for short or long precision respectively; the results returned are similarly normalized.

MATHEMATICAL FUNCTIONS

The library provides subroutines to deal with all float arithmetic generic functions and has separate modules for short and long precision real arguments, and also for short and long precision complex arguments where these are admissible.

Linkage to all mathematical subroutines is by means of the operating system standard.

Where evaluation or conversion of an argument is necessary, this is done prior to the invocation of the library module. Hence, all arguments passed to the mathematical subroutines must be of scale FLOAT. As such, it is assumed that the arguments are normalized in aligned fullword or doubleword fields for short or long precision respectively. The results returned are normalized similarly.

Real Arguments		
Function	Short float	Long float
SQRT	IHESQS	IHESQL
EXP	IHEEXS	IHEEXL
LOG,LOG2,LOG10	IHELNS	IHELNL
SIN, COS,SIND,COSD	IHESNS	IHESNL
TAN, TAND	IHETNS	IHETNL
ATAN, ATAND	IHEATS	IHEATL
SINH, COSH	IHESHS	IHESHL
TANH	IHETHS	IHETHL
ATANH	IHEHTS	IHEHTL
ERF, ERFC	IHEEFS	IHEEFL

Complex Arguments		
Function	Short float	Long float
SQRT	IHESQW	IHESQZ
EXP	IHEEXW	IHEEXZ
LOG	IHELNW	IHELNZ
SIN,COS,SINH,COSH	IHESNW	IHESNZ
TAN, TANH	IHETNW	IHETNZ
ATAN, ATANH	IHEATW	IHEATZ

Figure 46. Mathematical Functions

ARRAY FUNCTIONS

The library provides support for compiled code in the implementation of the PL/I array built-in functions SUM, PROD, POLY, ALL, and ANY. Calls to array function modules are by means of the operating system standard; the indexing routines, which are used internally by the library, use the PL/I standard calling sequence.

In all cases, the source arguments are arrays and the function value returned is a scalar. The evaluation of this function value requires only one call from compiled code, indexing through the array being handled internally within the library.

In the interests of efficiency, two sets of modules are provided: those which deal with arrays whose elements are stored contiguously (simple arrays), and those which also deal with arrays whose elements are not in contiguous storage (interleaved arrays).

In order to deal with array element addressing, the library modules require an array dope vector (ADV or SADV) to be passed as an argument. The format of these dope vectors is described in Appendix H. The number n, the number of dimensions of the array, is required in addition to the ADV or SADV, and is passed as a separate argument.

The PL/I language requires that the scalar values resulting from the use of the array functions, SUM, PROD, and POLY, should be floating-point. Since the library modules are addressing each array element successively, the necessary calls to the conversion routines (to change scale from FIXED to FLOAT) are made from the SUM, PROD, and POLY modules which have fixed-point arguments. In the case of ALL and ANY functions, it is expected that any necessary conversion to bit string will be carried out before the library is invoked.

	Simple arrays, and interleaved arrays of variable-length strings	Interleaved string arrays with fixed-length elements
Indexers ALL, ANY	IHEJXS IHENL1	IHEJXI IHENL2

Note: IHEJXI is also used for indexing through interleaved arithmetic arrays

PL/I functions		Fixed - point arguments		Floating-point arguments			
				Short precision		Long precision	
		Simple	Interleaved	Simple	Interleaved	Simple	Interleaved
SUM	real	IHESSF	IHESMF	IHESSG	IHESMG	IHESSH	IHESMH
	Complex	IHESSX	IHESMX	IHESSG	IHESMG	IHESSH	IHESMH
PROD	real	IHEPSF	IHEPDF	IHEPSS	IHEPDS	IHEPSL	IHEPDL
	complex	IHEPSX	IHEPDX	IHEPSW	IHEPDW	IHEPSZ	IHEPDZ
POLY	real	IHEYGF		IHEYGS		IHEYGL	
	complex	IHEYGX		IHEYGW		IHEYGZ	

Figure 47. Array Indexers and Functions

CHAPTER 9: MODULE SUMMARIES

This section provides information about individual modules of the PL/I Library. It serves as an introduction to the more detailed accounts given in the prefaces to the program listings. A brief statement of function is given; also provided are full specifications of linkage and inter-modular dependency. Since many library modules invoke the execution error package (IHEERR), no reference is made to this module in the 'Calls' section. Appendix G gives the lengths of the modules and indicates their locations (SYS1.PL1LIB or SYS1.LINKLIB).

CONTROL PROGRAM INTERFACES

The 'Calls' and 'Called by' sections include the use of the LINK and XCTL macros to pass control.

DATA PROCESSING

All integral values specified in the 'Linkage' section of the module description will be represented internally as fullword binary integers. Target fields will also be fullwords unless otherwise specified or implied (for example, for long floating-point results).

When FIXED data is passed to the library, a DED is associated with it in the linkage. In cases where the DED is not interrogated, the appropriate entry in the 'Linkage' section is marked with an asterisk.

Complex arguments are assumed to have real and imaginary parts stored next to each other in that order, so that the address of the real part suffices for both of them. Both parts are described by the same DED.

I/O Editing and Data Conversions

Target fields may, if desired, be overlapped with source fields in all cases except IHEVSA, IHEVSB, IHEVSC, IHEVSD, IHEVSE, and IHEVSF.

Strings: A source string field may coincide with a target string field in the modules listed in Figure 48. It should be noted that use of the same address for the dope vectors of source string and target string is not generally permitted, even though the string fields themselves may be overlapped. The exceptions to this are the entry points IHEBSKK and IHECSKK, where a considerable saving of time can be obtained by using the same address for both the first source and target SDVs.

Module	Source/target coincidence	
	First source field only	Either source field
IHEBSA	Yes	-
IHEBSO	-	Yes
IHEBSK	Yes	-
IHEBSM	Yes	-
IHEBSF	-	Yes
IHECSK	Yes	-
IHECSM	Yes	-

Figure 48. Coincidence of Source and Target Fields in some String Modules

The first byte of the result produced by the comparison modules IHEBSC, IHEBSD, and IHECSC contains:

Bits	Contents
0 to 1	Instruction length code 01
2 to 3	Condition code as below
4 to 7	Program mask (calling routine)

The condition code is set as follows :

00	Strings equal
01	First string compares low at first inequality
10	First string compares high at first inequality

Arithmetic: Target fields may, if desired, be overlapped with source fields in all cases except IHEXIU, IHEXIV, IHEXIW, IHEXIZ, IHEXXL and IHEXXS.

Mathematical: Target fields may, if desired, be overlapped with source fields in all cases except IHEEFL, IHEEFS, IHELNW, IHELNZ, IHESQW and IHESQZ.

MODULE SUMMARIES

IHEABU

ntry point: IHEABU0

Function:

ABS(z), where z is complex fixed-point binary.

Linkage:

```
RA: A(Parameter list)
Parameter list:
  A(z)
 *A(DED for z)
  A(Target)
 *A(Target DED)
```

Called by: Compiled code

IHEABV

Entry point: IHEABV0

Function:

ABS(z), where z is complex fixed-point decimal.

Linkage:

```
RA: A(Parameter list)
Parameter list:
  A(z)
  A(DED for z)
  A(Target)
  A(Target DED)
```

Called by: Compiled code

IHEABW

Calls: IHESQS

Entry point: IHEABW0

Function:

ABS(z), where z is complex short floating-point.

Linkage:

```
RA: A(Parameter list)
Parameter list:
  A(z)
  A(Target)
```

Called by: Compiled code, IHESQW

IHEABZ

Calls: IHESQL

Entry point: IHEABZ0

Function:

ABS(z), when z is complex long floating-point.

Linkage:

```
RA: A(Parameter list)
Parameter list:
  A(z)
  A(Target)
```

Called by: Compiled code, IHESQZ

IHEADD

Calls: IHEAPD

Entry point: IHEADD0

Function:

ADD(x,y,p,q), where x and y are real fixed-point decimal, and (p,q) is the target precision.

Linkage:

```
RA: A(Parameter list)
Parameter list:
  A(x)
  A(DED for x)
  A(y)
  A(DED for y)
  A(Target)
  A(Target DED)
```

Called by: Compiled code, IHEADV

IHEADV

Calls: IHEADD

Entry point: IHEADV0

Function:

ADD(w,z,p,q), where w and z are complex fixed-point decimal, and (p,q) is the target precision.

Linkage:

```
RA: A(Parameter list)
Parameter list:
  A(w)
  A(DED for w)
  A(z)
  A(DED for z)
  A(Target)
  A(Target DED)
```

Called by: Compiled code

IHEAPD

Entry point IHEAPDA

Function:

To assign x to a target with precision (p_2, q_2), where x is real fixed-point decimal with precision (p_1, q_1), and $p_1 \leq 31$.

Linkage:

RA: A(x)
RB: A(DED for x)
RC: A(Target)
RD: A(DED for target)

Called by: IHEADD, IHEDVV, IHEMPV

Entry point IHEAPDB

Function:

To convert x to precision $(31,q_2)$, where x is real fixed-point decimal with precision (p_1, q_1), and $p_1 \leq 31$.

Linkage: As for IHEAPDA

Called by: IHEADD, IHEDDV

IHEATL

Entry point IHEATL1

Function:

ATAN(x), where x is real long floating-point.

Linkage:

RA: A(Parameter list)
Parameter list:
  A(x)
  A(Target)

Called by: Compiled code

Entry point IHEATL2

Function:

ATAN(y,x), where x and y are real long floating-point.

Linkage:

RA: A(Parameter list)
Parameter list:
  A(y)
  A(x)
  A(Target)

Called by: Compiled code, IHEATZ, IHELNZ

Entry point IHEATL3

Function:

ATAND(x), where x is real long floating-point.

Linkage:

RA: A(Parameter list)
Parameter list:
  A(x)
  A(Target)

Called by: Compiled code

Entry point IHEATL4

Function:

ATAND(y,x), where x and y are real long floating-point.

Linkage:

RA: A(Parameter list)
Parameter list:
  A(y)
  A(x)
  A(Target)

Called by: Compiled code

IHEATS

Entry point IHEATS1

Function:

ATAN(x), where x is real short floating-point.

Linkage:

RA: A(Parameter list)
Parameter list:
  A(x)
  A(Target)

Called by: Compiled code

Entry point IHEATS2

Function:

ATAN(y,x), where x and y are real short floating-point.

Linkage:

RA: A(Parameter list)
Parameter list:
  A(y)
  A(x)
  A(Target)

Called by: Compiled code, IHEATW, IHELNW

Entry point IHEATS3

Function:

ATAND(x), where x is real short floating-point.

Linkage:

RA: A(Parameter list)
Parameter list:
A(x)
A(Target)

Called by: Compiled code

Entry point IHEATS4

Function:

ATAND(y,x), where x and y are real short floating-point.

Linkage:

RA: A(Parameter list)
Parameter list:
A(y)
A(x)
A(Target)

Called by: Compiled code

IHEATW

Calls: IHEATS, IHEHTS

Entry point IHEATWN

Function:

ATAN(z), where z is complex short floating-point.

Linkage:

RA: A(Parameter list)
Parameter list:
A(z)
A(Target)

Called by: Compiled code

Entry point IHEATWH

Function:

ATANH(z), where z is complex short floating-point.

Linkage:

RA: A(Parameter list)
Parameter list:
A(z)
A(Target)

Called by: Compiled code

IHEATZ

Calls: IHEATL,IHEHTL

Entry point IHEATZN

Function:

ATAN(z), where z is complex long floating-point.

Linkage:

RA: A (Parameter list)
Parameter list:
A (z)
A (Target)

Called by: Compiled code

Entry point IHEATZH

Function:

ATANH (z), when z is complex long floating-point.

Linkage:

RA: A (Parameter list)
Parameter list:
A (z)
A (Target)

Called by: Compiled code

IHEBEG

Calls:

Supervisor (LINK, GETMAIN, FREEMAIN), IHETOM

Entry point IHEBEGA

Function:

Links to IHETOM to issue a WTO macro insruction if the PRV is longer than 4096 bytes.

Linkage: None

Called by: IHESA, IHETSA

Entry point IHEBEGN

Function:

Links to IHETOM to issue a WTO macro instruction if the program does not have a main procedure.

Linkage: None

Called by: IHESA, IHETSA

IHEBSA

Entry point: IHEBSA0

Function:

AND operator (&) for two byte-aligned bit strings.

Linkage:

```
RA: A(SDV of first operand)
RB: A(SDV of second operand)
RC: A(SDV of target field)
```

Called by: Compiled code, IHENL1, IHENL2

IHEBSC

Entry point: IHEBSC0

Function:

To compare two byte-aligned bit strings.

Linkage:

```
RA: A(SDV of first operand)
RB: A(SDV of second operand)
RC: A(Target)
```

Called by: Compiled code

IHEBSD

Entry point: IHEBSD0

Function:

To compare two bit strings with any alignment.

Linkage:

```
RA: A(SDV of first operand)
RB: A(SDV of second operand)
RC: A(Target)
```

Called by: Compiled code

IHEBSF

Entry point: IHEBSF0

Function:

BOOL (Bit string, bit string, string n_1 n_2 n_3 n_4).

Linkage:

RA: A(Parameter list)
Parameter list:
  A(SDV of first source string)
  A(SDV of second source string)
  A(Fullword containing bit pattern n_1 n_2 n_3 n_4 right justified)
  A(SDV of target field)

Called by: Compiled code,IHENL1,IHENL2

IHEBSI

Entry point: IHEBSI0

Function:

INDEX (Bit string, bit string).

Linkage:

```
RA: A(Parameter list)
Parameter list:
  A(SDV of first source string)
  A(SDV of second source string)
  A(Target field)
```

Called by: Compiled code

IHEBSK

Entry point IHEBSKA

Function:

To assign a bit string to a target field.

Linkage:

```
RA: A(SDV of source string)
RB: A(SDV of target field)
```

Called by: Compiled code

Entry point IHEBSKK

Function:

Concatenate operator (||) for bit strings.

Linkage:

```
RA: A(SDV of first operand)
RB: A(SDV of second operand)
RC: A(SDV of target field)
```

Called by: Compiled code

Entry point IHEBSKR

Function: REPEAT (Bit string,n).

Linkage:

RA: A(SDV of source string)

RB: A(n)
RC: A(SDV of target field)

Called by: Compiled code

IHEBSM

Entry point IHEBSMF

Function:

To assign a byte-aligned bit string to a byte-aligned fixed-length target.

Linkage:

RA: A(SDV of source string)
RB: A(SDV of target field)

Called by: Compiled code

Entry point IHEBSMV

Function:

To assign a byte-aligned bit string to a byte-aligned VARYING target.

Linkage: As for IHEBSMF

Called by: Compiled code

Entry point IHEBSMZ

Function:

To fill out a bit string from its current length to its maximum length with zero bits.

Linkage: RA: A(SDV)

Called by: Compiled code

IHEBSN

Entry point: IHEBSN0

Function:

NOT operator (¬) for a byte-aligned bit string.

Linkage:

RA: A(SDV of operand)
RB: A(SDV of target field)

Called by: Compiled code

IHEBSO

Entry point: IHEBSO0

Function:

OR operator (|) for two byte-aligned bit strings.

Linkage:

RA: A(SDV of first operand)
RB: A(SDV of second operand)
RC: A(SDV of target field)

Called by: Compiled code, IHENL1, IHENL2

IHEBSS

Entry point IHEBSS2

Function:

To produce an SDV describing the pseudo-variable or function SUBSTR (Bit string, i).

Linkage:

RA: A(Parameter list)
Parameter list:
  A(SDV of source string)
  A(i)
  Dummy argument
  A(Field for target SDV)

Called by: Compiled code

Entry point IHEBSS3

Function:

To produce an SDV describing the pseudo-variable or function SUBSTR (Bit string, i, j).

Linkage:

RA:A(Parameter list)
Parameter list:
  A(SDV of source string)
  A(i)
  A(j)
  A(Field for target SDV)

Called by: Compiled code

IHECFA

Entry point: IHECFAA

Function:

ONLOC: Locates the BCD name of the procedure that contains the PL/I interrupt that caused entry into the current on-unit. If ONLOC is specified outside an on-unit, a null string is returned.

Linkage:

RA: A(Parameter list)
Parameter list: A(Target SDV)

Called by: Compiled code

IHECFB

Entry point: IHECFBA

Function:

ONCODE: Returns a value corresponding to the condition which caused the interrupt. If specified outside an on-unit, a unique code (0) is returned.

Linkage:

```
RA: A(Parameter list)
Parameter list:
  A(4-byte word-aligned target)
```

Called by: Compiled code

IHECFC

Entry point: IHECFCA

Function:

ONCOUNT: Returns a value equal to the number of PL/I conditions and program exceptions, including the current one, that have yet to be processed. A zero value is returned if:

1. ONCOUNT is used outside an ON unit, or

2. ONCOUNT is used in an ON unit entered because of a precise interrupt or a single imprecise interrupt

(This built-in function is used in connection with the Model 91 option)

Linkage:

```
RA: A(Parameter list)
Parameter list:
  A(4-byte word-aligned target)
```

Called by: Compiled code

IHECKP

Calls: Supervisor

Entry point: IHECKPT

Function:

To call the control program checkpoint facility to save main storage areas and control information so that the job step may be restarted from the checkpoint.

Linkage: None

Called by:

Compiled code(CALL IHECKPT statement)

IHECLT

Calls:

IHESA, Supervisor (CLOSE, DCBD, DELETE, FREEMAIN, FREEPOOL, RETURN)

Entry point IHECLTA

Function:

Close files:

1. Free FCB.

2. Set file register to zero.

3. Remove file from IHEQFOP chain.

4. Delete interface modules loaded for record-oriented I/O.

5. Purge outstanding I/O events, setting event variables complete, abnormal, and inactive.

Linkage:

```
RA: A(Parameter list)
Parameter list:
  A(CLOSE parameter list)
  A(Private adcons)

CLOSE parameter list:
  A(DCLCB1)
  (Reserved)
  (Reserved)
  .
  .
  .
  A(DCLCBn)
  (Reserved)
  (Reserved)
  (High-order byte of last argument
  indicates end of parameter list)
```

Called by: IHEOCL

Entry point IHECLTB

Function:

To close all files when a task is terminated.

Linkage:

```
RA: A(Parameter list)
Parameter list:
    F(number of files to be closed*4)
    A(Adcon list)
    A(1st FCB)
    .
    .
    A(nth FCB)
    (High-order byte of last argument
    indicates end of parameter list.)
```

Called by: IHEOCL

IHECNT

Entry point IHECNTA

Function:

Returns count of scalar items transmitted on last I/O operation.

Linkage:

RA: A(Parameter list)
Parameter list:
  A(DCLCB)
  A(Fullword)

Entry point IHECNTB

Function:

Returns current line number (LINENO).

Linkage: As for IHECNTA

IHECSC

Entry point: IHECSC0

Function:

To compare two character strings.

Linkage:

RA: A(SDV of first operand)
RB: A(SDV of second operand)
RC: A(Target)

Called by: Compiled code

IHECSI

Entry point: IHECSI0

Function:

INDEX (Character string, character string).

Linkage:

RA: A(Parameter list)
Parameter list:
  A(SDV of first source string)
  A(SDV of second source string)
  A(Target field)

Called by: Compiled code

IHECSK

Entry point IHECSKK

Function:

Concatenate operator (||) for character strings.

Linkage:

RA: A(SDV of first operand)
RB: A(SDV of second operand)
RC: A(SDV of target field)

Called by: Compiled code

Entry point IHECSKR

Function:

REPEAT (Character string, n).

Linkage:

RA: A(SDV of source string)
RB: A(n)
RC: A(SDV of target field)

Called by: Compiled code

IHECSM

Entry point IHECSMF

Function:

To assign a character string to a fixed-length target.

Linkage:

RA: A(SDV of source string)
RB: A(SDV of target field)

Called by: Compiled code

Entry point IHECSMV

Function:

To assign a character string to a VARYING target.
Linkage: As for IHECSMF

Called by: Compiled code

Entry point IHECSMB

Function:

To fill out a character string from its current length to its maximum length with blanks.

Linkage:

RA: A(SDV)

Called by: Compiled code

Entry point IHECSMH

Function: HIGH

Linkage: As for IHECSMB

Called by: Compiled code

Entry point IHECSML

Function: LOW.

Linkage: As for IHECSMB

Called by: Compiled code

IHECSS

Entry point IHECSS2

Function:

To produce an SDV describing the pseudo-variable or function SUBSTR (Character string, i).

Linkage:

```
RA: A(Parameter list)
Parameter list:
  A(SDV of source string)
  A(i)
  Dummy argument
  A(Field for target SDV)
```

Called by: Compiled code

Entry point IHECSS3

Function:

To produce an SDV describing the pseudo-variable or function SUBSTR (Character string, i, j).

Linkage:

```
RA: A(Parameter list)
Parameter list:
  A(SDV of source string)
  A(i)
  A(j)
  A(Field for target SDV)
```

Called by: Compiled code

IHECTT

Calls:

IHETSA, Supervisor (CLOSE, DCBD, DELETE, DEQ, FREEMAIN, FREEPOOL, RETURN)

Entry point IHECTTA

Function:

Close files in a multitasking environment:

1. Free FCB.
2. Set file register to zero.
3. Remove file from IHEQFOP chain.
4. Delete interface modules loaded for record-oriented I/O.
5. Purge outstanding I/O events, setting event variables complete, normal, and inactive.

   (i) Check that the file is in the IHEQFOP chain for the current task.

   (ii) Free IOCBs, setting associated EVENT variables complete, abnormal, and inactive.

   (iii) Set EVENT variables in TEVT chain complete, abnormal, and inactive.

   (iv) For REGIONAL EXCLUSIVE files, or INDEXED EXCLUSIVE files with unblocked records, dequeue locked records and free EXCLUSIVE blocks in the TXLV chain.

   (v) For INDEXED EXCLUSIVE files with blocked records, unlock the files.

Linkage:

```
RA: A(Parameter list)
Parameter list:
  A(CLOSE parameter list)
  A(Private adcons)

CLOSE parameter list:
  A(DCLCB₁)
  A(IDENT SDV₁)/0
  A(IDENT DED₁)/0
  .
  .
  .
  A(DCLCBₙ)
  A(IDENT SDVₙ)/0
  A(IDENT DEDₙ)/0
  (High-order byte of last argument
  indicates end of parameter list)
```

Called by: IHEOCT

Entry point IHECTTB

Function:

To close all files when a task is terminated.

Linkage:

```
RA: A(Parameter list)
Parameter list:
  F(number of files to be closed*4)
  A(Adcon list)
  A(1st FCB)
  .
  .
  A(nth FCB)
  (High-order byte of last argument
  indicates end of parameter list)
```

Called by: IHEOCT

IHEDBN

Calls: IHEDMA, IHEUPA, IHEUPB

Entry point: IHEDBNA

Function:

To convert a bit string to an arithmetic target with a specified base, scale, mode, and precision.

Linkage:

```
RA: A(Source SDV)
RB: A(Source DED)
RC: A(Target)
RD: A(Target DED)
```

Called by:

Compiled code, IHEDID, IHEDOA, IHEDOE, IHEDOM

IHEDCN

Calls: IHEDMA, IHEUPA, IHEUPB, IHEVQB

Entry point IHEDCNA

Function:

To convert a character string containing a valid arithmetic constant or complex expression to an arithmetic target with specified base, scale, mode, and precision. The ONSOURCE address is stored.

Linkage:

```
RA: A(Source SDV)
RB: A(Source DED)
RC: A(Target)
RD: A(Target DED)
WOFD: A(Source SDV)
```

Called by:

Compiled code, IHEDIB, IHEDOE, IHELDI

Entry point IHEDCNB

Function:

As for IHEDCNA, but the ONSOURCE address is not stored.

Linkage:

As for IHEDCNA, but without WOFD

Called by: As for IHEDCNA

IHEDDI

Calls:

IHEDDJ, IHEIOF, IHELDI, IHESA, IHETSA

Entry point IHEDDIA

Function:

To read data from an input stream and assign it to internal variables according to symbol table information conventions. Restrictive data list.

Linkage:

```
RA: A(Parameter list)
Parameter list:
  A(Symbol table₁)
    .
    .
    .
  A(Symbol tableₙ)
  (High-order byte of last argument
   indicates end of parameter list.)
```

Called by: Compiled code

Entry point IHEDDIB

Function:

As for IHEDDIA, but no data list.

Linkage:

```
RA: A(Parameter list)
Parameter list: A(Symbol table chain)
```

Called by: Compiled code

IHEDDJ

Entry point: IHEDDJA

Function:

To compute the address of an array element from source subscripts and an ADV.

Linkage:

RA: A(ADV)
RB: A(DED)
RC: A(Field for element address)
RD: A(Symbol table entry, 2nd part)
RE: A(SDV for subscripts)

Called by: IHEDDI

IHEDDO

Calls:

IHEDDP, IHEIOF, IHELDO, IHEPRT

Entry point IHEDDOA

Function:

To convert data according to data-directed output conventions and to write it onto an output stream. For scalar variables and whole arrays.

Linkage:

RA: A(Parameter list)
Parameter list:
A(Symbol table entry$_1$)
.
.
.
A(Symbol table entry$_n$)
(High-order byte of last argument indicates end of parameter list.)

Called by: Compiled code

Entry point IHEDDOB

Function:

As for IHEDDOA but for array variable elements.

Linkage:

RA: A(Parameter list)
Parameter list:
A(Symbol table entry$_1$)
A(Element address$_1$)
.
.
.
A(Symbol table entry$_n$)
A(Element address$_n$)
(High-order byte of last argument indicates end of parameter list.)

Called by: Compiled code

Entry point IHEDDOC

Function:

To terminate data-directed transmission.

Linkage: None

Called by: Compiled code

Entry point IHEDDOD

Function:

As for IHEDDOA, but used to support the CHECK condition.

Linkage:

RA: A(Parameter list)
Parameter list:
A(Symbol table entry)
A(Element address)

Called by: IHEERR

Entry point IHEDDOE

Function:

In the absence of a data list, to convert all data known within a block according to data-directed output conventions and to write it onto an output stream.

Linkage:

RA: A(Parameter list)
Parameter list:
A(First symbol table entry)

Called by: Compiled code

IHEDDP

Calls: IHEERR

Entry point IHEDDPA

Function:

To prepare an array for subscript output operation, and to address the first element.

Linkage:

RA: A(Field for A(VDA))
RB: A(FCB)
RC: A(Symbol table entry, 2nd part)

Called by: IHEDDO

Entry point IHEDDPB

Function: To perform subscript output.

Linkage:

RA: A(Parameter list)
Parameter list: A(VDA)

Called by: IHEDDO

Entry point IHEDDPC

Function: To address the next element.

Linkage:

RA: A(Parameter list)
Parameter list: A(VDA)
Return codes:
BR=0: Another element
BR=4: End of array

Called by: IHEDDO

Entry point IHEDDPD

Function:

To prepare an array for subscript output operation for a given element.

Linkage:

RA: A(Field for A(VDA))
RB: A(FCB)
RC: A(Symbol table entry, 2nd part)
RD: A(Element)

Called by: IHEDDO

IHEDDT

Calls:

Supervisor (DEQ,ENQ), IHEDDP, IHEIOF, IHELDO, IHEPTT

Entry point IHEDDTA

Function:

To convert data according to data-directed output conventions and to write it onto an output stream. For scalar variables and whole arrays in a multitasking environment.

Linkage:

RA: A(Parameter list)
Parameter list:
A(Symbol table entry$_1$)
.
.
.
A(Symbol table entry$_n$)
(High-order byte of last argument indicates end of parameter list)

Called by: Compiled code

Entry point IHEDDTB

Function:

As for IHEDDTA but for array variable elements.

Linkage:

RA: A(Parameter list)
Parameter list:
A(Symbol table entry$_1$)
A(Element address$_1$)
.
.
.
A(Symbol table entry$_n$)
A(Element address$_n$)
(High-order byte of last argument indicates end of parameter list)

Called by: Compiled code

Entry point IHEDDTC

Function:

To terminate data-directed transmission in a multitasking environment.

Linkage: None

Called by: Compiled code

Entry point IHEDDTD

Function:

As for IHEDDTA, but used to support the CHECK condition in a multitasking environment.

Linkage:

RA: A(Parameter list)
Parameter list:
A(Symbol table entry)
A(Element address)

Called by: IHEERR

Entry point IHEDDTE

Function:

In the absence of a data list, to convert all data known within a block according to data-directed output conventions and to write it onto an output stream in a multitasking environment.

Linkage:

RA: A(Parameter list)
Parameter list:
A(First symbol table entry)

Called by: Compiled code

IHEDIA

Calls:

IHEDMA, IHEDNB, IHEDNC, IHEIOD, IHEUPA, IHEUPB, IHEVCA, IHEVSA, IHEVSC, IHEVQB

Entry point IHEDIAA

Function:

To direct the conversion of F format data to an internal data type.

Linkage:

RA: A(Target or target dope vector)
RB: A(Target DED)
RC: A(FED)

Called by: Compiled code, IHEDIM

Entry point IHEDIAB

Function:

To direct the conversion of E format data to an internal data type.

Linkage: As for IHEDIAA

Called by: As for IHEDIAA

IHEDIB

Calls:

IHEDCN, IHEIOD, IHEKCD, IHEVSC, IHEVSD, IHEVSE

Entry point IHEDIBA

Function:

To direct the conversion of A format data to an internal data type.

Linkage:

RA: A(Target or target dope vector)
RB: A(Target DED)
RC: A(FED)

Called by: Compiled code

Entry point IHEDIBB

Function:

To direct the conversion of pictured character string data to an internal data type.

Linkage: As for IHEDIBA

Called by: Compiled code

IHEDID

Calls:

IHEDMA, IHEIOD, IHEUPA, IHEUPB, IHEVSC, IHEVSD, IHEVSE

Entry point: IHEDIDA

Function:

To direct the conversion of external B format data to an internal data type.

Linkage:

RA: A(Target or target dope vector)
RB: A(Target DED)
RC: A(FED)

Called by: Compiled code

IHEDIE

Calls:

IHEDMA, IHEIOD, IHEKCA, IHEKCB, IHEUPA, IHEUPB, IHEVSC, IHEVSD, IHEVSE

Entry point: IHEDIEA

Function:

To direct the conversion of external data with a numeric picture format to an internal data type.

Linkage:

RA: A(Target or target dope vector)
RB: A(Target DED)
RC: A(FED)

Called by: Compiled code, IHEDIM

IHEDIL

Entry point IHEDILA

Function:

To set up appropriate error handling when no width specification for A format input is given.

Linkage: None

Called by: Compiled code

Entry point IHEDILB

Function:

As for IHEDILA, but B format

Linkage: None

Called by: Compiled code

IHEDIM

Calls:

IHEDIA, IHEDIE, IHEIOD, IHEKCA, IHEVCA, IHEVCS

Entry point: IHEDIMA

Function:

To direct the conversion of external data with C format to an internal data type.

Linkage:

RA: A(Target or target dope vector)
RB: A(Target DED)
RC: A(Real format director)
RD: A(Real FED)
RE: A(Imaginary format director)
RF: A(Imaginary FED)

Called by: Compiled code

IHEDMA

Transfers control to:

IHEVFD, IHEVFE, IHEVKB, IHEVKC, IHEVPE, IHEVPF, IHEVPG, IHEVPH

Entry point: IHEDMAA

Function:

To set up the intermodular flow to effect conversion from one arithmetic data type to another.

Linkage:

RA: A(Source)
RB: A(Source DED)
RC: A(Target)
RD: A(Target DED)

Called by:

Compiled code, I/O directors, IHEDBN, IHEDCN, IHEDNB, IHEDNC, IHELDI, IHEPDF, IHEPDX, IHEPSF, IHEPSX, IHESMF, IHESMX, IHESSF, IHEUPB, IHEVCS, IHEYGF, IHEYGX

IHEDNB

Calls: IHEDMA, IHEVSA

Entry point: IHEDNBA

Function:

To convert an arithmetic source with specified base, scale, mode, and precision to a fixed-length bit string or a VARYING bit string of specified length.

Linkage:

RA: A(Source)
RB: A(Source DED)
RC: A(Target SDV)
RD: A(Target DED)

Called by:

Compiled code, IHEDIA, IHEDOD, IHELDI, IHEVCS

IHEDNC

Calls:

IHEDMA, IHEUPA, IHEVSC, IHEVSE, IHEVQC

Entry point: IHEDNCA

Function:

To convert an arithmetic source of specified base, scale, mode, and precision to a character string or a pictured character string.

Linkage:

RA: A(Source)
RB: A(Source DED)
RC: A(Target SDV)
RD: A(Target DED)

Called by:

Compiled code, IHEDIA, IHEDOB, IHELDI, IHELDO, IHEVCS

IHEDOA

Calls:

IHEDBN, IHEDMA, IHEIOD, IHEVQC

Entry point IHEDOAA

Function:

To direct the conversion of internal data to external F format.

Linkage:

RA: A(Source or source dope vector)
RB: A(Source DED)
RC: A(FED)

Called by: Compiled code, IHEDOM

Entry point IHEDOAB

Function:

To direct the conversion of internal data to external E format.

Linkage: As for IHEDOAA

Called by: As for IHEDOAA

IHEDOB

Calls:

IHEDNC, IHEIOD, IHEVSB, IHEVSC, IHEVSE, IHEVSF

Entry point IHEDOBA

Function:

To direct the conversion of internal data to external A(w) format.

Linkage:

RA: A(Source or source dope vector)
RB: A(Source DED)
RC: A(FED)

Called by: Compiled code

Entry point IHEDOBB

Function:

To direct the conversion of internal data to external A format.

Linkage:

RA: A(Source or source dope vector)
RB: A(Source DED)

Called by: Compiled code

Entry point IHEDOBC

Function:

To direct the conversion of internal data to external pictured character format.

Linkage: As for IHEDOBA

Called by: Compiled code

IHEDOD

Calls: IHEDNB, IHEIOD, IHEVSB, IHEVSC

Entry point IHEDODA

Function:

To direct the conversion of internal data to external B(w) format.

Linkage:

RA: A(Source or source dope vector)
RB: A(Source DED)
RC: A(FED)

Called by: Compiled code

Entry point IHEDODB

Function:

To direct the conversion of internal data to external B format.

Linkage:

RA: A(Source or source dope vector)
RB: A(Source DED)

Called by: Compiled code

IHEDOE

Calls:

IHEDBN, IHEDCN, IHEDMA, IHEIOD, IHEVSB

Entry point: IHEDOEA

Function:

To direct the conversion of internal data to external data with a numeric picture format.

Linkage:

RA: A(Source or source dope vector)
RB: A(Source DED)
RC: A(FED)

Called by: Compiled code, IHEDOM

IHEDOM

Calls:

IHEDBN, IHEDOA, IHEDOE, IHEUPA, IHEUPB, IHEVCA, IHEVCS

Entry point: IHEDOMA

Function:

To direct the conversion of an internal data type to external C format data.

Linkage:

RA: A(Source or source dope vector)
RB: A(Source DED)
RC: A(Real format director)
RD: A(Real FED)
RE: A(Imaginary format director)
RF: A(Imaginary FED)

Called by: Compiled code

<u>IHEDSP</u>

Calls: Supervisor (WAIT, WTO, WTOR, GETMAIN, POST, FREEMAIN, CHAP)

Entry point: IHEDSPA

Function:

To write a message on the operator's console, with or without a reply. The EVENT option can be used for a message with a reply.

Linkage:

RA: A(Parameter list)
Parameter list:
  A(SDV for message)
  A(SDV for reply)
  A(Event variable)
  (The parameter list is either one, two, or three elements long, depending on the use of the REPLY and EVENT options. The high-order byte of the last argument indicates the end of the parameter list.)

Called by: Compiled code

<u>IHEDUM</u>

Calls:

Supervisor (ABEND, SNAP), IHETSA, IHEZZC

<u>Entry point IHEDUMC</u>

Function:

Dump current task and then continue execution.

Linkage:

RA: A(Parameter list)
Parameter list:
  F(Number in range 0 through 255)

Called by:

Compiled code (CALL IHEDUMC statement)

<u>Entry point IHEDUMJ</u>

Function:

Dump all tasks and then continue execution.

Linkage: As IHEDUMC

Called by:

Compiled code (CALL IHEDUMJ statement)

<u>Entry point IHEDUMP</u>

Function:

Dump all tasks and terminate major task.

Linkage: As IHEDUMC

Called by:

Compiled code (CALL IHEDUMP statement)

<u>Entry point IHEDUMT</u>

Function:

Dump current task and then terminate it.

Linkage: As IHEDUMC

Called by:

Compiled code (CALL IHEDUMT statement)

<u>IHEDVU</u>

Entry point: IHEDVU0

Function:

DIVIDE(w,z,p,q), where w and z are complex fixed-point binary, and (p,q) is the target precision.

Linkage:

RA: A(Parameter list)
Parameter list:
  A(w)
  A(DED for w)
  A(z)
  A(DED for z)
  A(Target)
  A(DED for target)

Called by : Compiled code

<u>IHEDVV</u>

Calls: IHEAPD

Entry point: IHEDVV0

Function:

DIVIDE(w,z,p,q), where w and z are complex fixed-point decimal, and (p,q) is the target precision.

Linkage:

RA: A(Parameter list)
Parameter list:
A(w)
A(DED for w)
A(z)
A(DED for z)
A(Target)
A(DED for target)

Called by: Compiled code

IHEDZW

Entry point: IHEDZW0

Function:

z_1/z_2, where z_1 and z_2 are complex short floating-point.

Linkage:

RA: A(z_1)
RB: A(z_2)
RC: A(Target)

Called by: Compiled code

IHEDZZ

Entry point: IHEDZZ0

Function:

z_1/z_2, where z_1 and z_2 are complex long floating-point.

Linkage:

RA: A(z_1)
RB: A(z_2)
RC: A(Target)

Called by: Compiled code

IHEEFL

Calls: IHEEXL

Entry point IHEEFLF

Function:

ERF(x), where x is real long floating-point.

Linkage:

RA: A(Parameter list)
Parameter list:
A(x)
A(Target)

Called by: Compiled code

Entry point IHEEFLC

Function:

ERFC(x), where x is real long floating-point.

Linkage: As for IHEEFLF

Called by: Compiled code

IHEEFS

Calls: IHEEXS

Entry point IHEEFSF

Function:

ERF(x), where x is real short floating-point.

Linkage:

RA: A(Parameter list)
Parameter list:
A(x)
A(Target)

Called by: Compiled code

Entry point IHEEFSC

Function:

ERFC(x), where x is real short floating-point.

Linkage: As for IHEEFSF

Called by: Compiled code

IHEERD

Function:

Non-resident part of the error-handling routines. It contains the data-processing error messages, and when required is dynamically loaded from IHEESM (Versions 3 and 4).

IHEERE

Function:

Non-resident part of the error-handling routines. It contains the input/output error messages, and when required is dynamically loaded from IHEESM (Versions 3 and 4).

IHEERI

Function:

Non-resident part of the error-handling routines. It contains the remaining error messages, that is, those not contained in IHEERD, IHEERE, IHEERO and IHEERP, and when required is dynamically loaded from IHEESM (Versions 3 and 4).

IHEERN

Function:

Non-resident part of the error package. It contains the error messages, and is dynamically loaded as required by IHEERR (Version 1) or IHEESS (Version 2).

IHEERO

Function:

Non-resident part of the error-handling routines. It contains the error messages, and when required is dynamically loaded from IHEESM (Versions 3 and 4).

IHEERP

Function:

Non-resident part of the error-handling routines. It contains the error messages, and when required is dynamically loaded from IHEESM (Versions 3 and 4).

IHEERR

Calls:

Supervisor (LINK, SPIE), IHEDDO, IHEDDT, IHEERS (Version 1), IHEESM, IHEESS (Version 2), IHEM91, IHEPRT, IHEPTT, IHESA, IHETER, IHETSA
IHEERRE calls: LINK, ABEND with DUMP and STEP options

Entry point IHEERRA (Program Interrupt):

Function:

To determine the identity of the error or condition that has been raised, and to determine what action must be taken on account of it. Several courses of action are possible, including combinations of:

(1) Entry into an on-unit
(2) SNAP
(3) No action - return to program
(4) Print error message and terminate
(5) Print error message and continue
(6) Set standard results into float registers

Linkage: None

Called by: Supervisor

Entry point IHEERRB (ON Conditions):

Function: As for IHEERRA.

Linkage:

RA: A(DCLCB)(for I/O conditions)
IHEQERR: Error code

Called by: Compiled code, library modules

Entry point IHEERRC (Non-ON errors):

Function: As for IHEERRA.

Linkage:

RA: A(Two-byte error code)
A(Four-byte code if source program error)

Called by: Compiled code, library modules

Entry point IHEERRD (CHECK, CONDITION):

Function: As for IHEERRA.

Linkage:

RA: A(Parameter list)
Parameter list:
One-byte error code
Three-byte A(X)
X: Symbol table

X: Symbol table (CHECK variable), or
Symbol length and name(CHECK label), or
Identifying CSECT(CONDITION)

Called by: Compiled code

Entry point IHEERRE

Function:

To accept control when a program interrupt occurs in IHEERR or in modules that IHEERR calls or links to; to link to IHETOM to write a disaster message on the console; to terminate the program and to provide an operating system ABDUMP.

Linkage: None

Called by: Supervisor

IHEERS

Entry point: IHEERSA

Function:

SNAP: To determine and record the location of the point of interrupt and to print the procedure trace-back information associated with it.

Linkage:

RA: A(Third word of a library VDA to be used as a save area and message buffer): words 21 to 23 of the VDA are used to pass the following parameters:
21: A(Interrupt VDA)/0
22: A(PRINT routine)
23: A(Current DSA)

Called by: IHEERR (Version 1)

IHEERT

Function:

Non-resident part of the error-handling routines. It contains the multitasking error messages and is dynamically loaded when required from IHEESM or IHETEX (Version 4).

IHEESM

Calls:

Supervisor (DELETE, DEQ, ENQ, LOAD), IHEERD, IHEERE, IHEERI, IHEERO, IHEERP, IHEERT, IHEPRT, IHEPTT, IHESA, IHETSA

Entry point IHEESMA

Function:

To print out SNAP and system action messages.

Linkage:

RA: A(First word of a library VDA to be used as a save area and message buffer)

RH: A(Current DSA)

Also passed are:
A(IHEPTTB) or A(IHEPRTB): current LWE + 124
A(IHETSAL) or A(IHESADE): current LWE + 128
A(IHETSAF) or A(IHESAFD): current LWE + 132
Length of PRV: current LWE+102

Called by: IHEERR (Versions 3 and 4)

Entry point IHEESMB

Function:

To print CHECK (label) system action messages.

Linkage:

RA: A(Label)
RB: A(Length of label)

Also passed:
A(IHEPTTB) or A(IHEPRTB): Current LWE + 124

Called by: IHEERR (Versions 3 and 4)

IHEESS

Calls: IHEERN, IHEPRT, IHESA, IHETSA

Entry point IHEESSA

Function:

To print out SNAP and system action messages.

Linkage:

RA: A(First word of a library VDA to be used as a save area and message buffer)

Also passed are:

A(Interrupt VDA/0):	current LWE + 96
A(Current DSA):	current LWE + 100
A(IHESADE):	current LWE + 104
A(IHESAFE):	current LWE + 108
A(IHEPRT):	current LWE + 112

Called by: IHEERR (Version 2)

Entry point IHEESSB

Function:

To print CHECK (label) system action messages.

Linkage:

RA: A(Label)
A(Length of label)

Also passed:

A(IHEPRTB): current LWE + 112

Called by: IHEERR (Version 2)

IHEEXL

Entry point: IHEEXL0

Function:

EXP(x), where x is real long floating-point.

Linkage:

```
RA: A(Parameter list)
Parameter list:
  A(x)
  A(Target)
```

Called by:

Compiled code, IHEEFL, IHEEXZ, IHESHL, IHESNZ, IHETHL, IHEXXL

IHEEXS

Entry point: IHEEXS0

Function:

EXP(x), where x is real short floating-point.

Linkage:

```
RA: A(Parameter list)
Parameter list:
  A(x)
  A(Target)
```

Called by:

Compiled code, IHEEFS, IHEEXW, IHESHS, IHESNW, IHETHS, IHEXXS

IHEEXW

Calls: IHEEXS, IHESNS

Entry point: IHEEXW0

Function:

EXP(z), where z is complex short floating-point.

Linkage:

```
RA: A(Parameter list)
Parameter list:
  A(z)
  A(Target)
```

Called by: Compiled code, IHEXXW

IHEEXZ

Calls: IHEEXL, IHESNL

Entry point: IHEEXZ0

Function:

EXP(z), where z is complex long floating-point.

Linkage:

```
RA: A(Parameter list)
Parameter list:
  A(z)
  A(Target)
```

Called by: Compiled code, IHEXXZ

IHEHTL

Calls: IHELNL

Entry point: IHEHTL0

Function:

ATANH(x), where x is real long floating-point.

Linkage:

```
RA: A(Parameter list)
Parameter list:
  A(x)
  A(Target)
```

Called by: Compiled code, IHEATZ

IHEHTS

Calls: IHELNS

Entry point: IHEHTS0

Function:

ATANH(x), where x is real short floating-point. point.

Linkage:

```
RA: A(Parameter list)
Parameter list:
  A(x)
  A(Target)
```

Called by: Compiled code, IHEATW

IHEIBT

This module is used in a multitasking environment and is equivalent to module IHEIOB in a non-multitasking environment.

Calls:

Supervisor (DEQ,ENQ), IHEIOP, IHEOCT

Entry point IHEIBTA

Function:

To initialize the PUT operation, and to check the file status, in a multitasking environment:

1. Open
2. Transmit error
3. Invalid

Linkage:

RA: A(Parameter list)
Parameter list:
  A(DCLCB)
  A(Abnormal return)

Called by: Compiled code

Entry point IHEIBTB

Function:

To initialize PUT, and perform PAGE, and to check the file status, in a multitasking environment:

1. Open
2. Transmit error
3. Invalid

Linkage: As for IHEIBTA

Called by: Compiled code

Entry point IHEIBTC

Function:

To initialize PUT, and perform SKIP, and to check the file status, in a multitasking environment:

1. Open
2. Transmit error
3. Invalid

Linkage:

RA: A(Parameter list)
Parameter list:
  A(DCLCB)
  A(Abnormal return)
  A(Expression value)

Called by: Compiled code

Entry point IHEIBTD

Function:

To initialize PUT, and perform LINE, and to check the file status, in a multitasking environment:

1. Open
2. Transmit error
3. Invalid

Linkage: As for IHEIBTC

Called by: Compiled code

Entry point IHEIBTE

Function:

To initialize PUT, and perform PAGE and LINE, and to check the file status, in a multitasking environment:

1. Open
2. Transmit error
3. Invalid

Linkage: As for IHEIBTC

Called by: Compiled code

Entry point IHEIBTT

Function:

To terminate the PUT operation, in a multitasking environment.

Linkage: None

Called by: Compiled code

IHEIGT

Entry point: IHEIGTA

Function:

As for IHEINT

IHEINT

This module is used in a multitasking environment and is equivalent to module IHEION in a non-multitasking environment.

Calls:

Supervisor(CHAP, FREEMAIN, GETMAIN), IHEITB, IHEITC, IHEITD, IHEITE, IHEITF, IHEITG, IHEITH, IHEITJ, IHEOCT

Entry point: IHEINTA

Function:

To verify a RECORD I/O request and to invoke the appropriate data management interface module to perform the required operation, in a multitasking environment.

Linkage:

RA: A(Parameter list)
Parameter list:
A(DCLCB)
A(RDV)/(IGNORE factor)
A(EVENT variable)/(0)/A(Error return)
A(KEY|KEYFROM|KEYTO SDV)/(0)
A(Request control block)

Called by: Compiled code

IHEIOA

Calls: IHEIOP, IHEOCL, IHEOCT

Entry point IHEIOAA

Function:

To initialize the GET operation, and to check the file status:

1. Open
2. Endfile
3. Invalid

Linkage:

RA: A(Parameter list)
Parameter list:
A(DCLCB)
A(Abnormal return)

Called by: Compiled code

Entry point IHEIOAB

Function:

To initialize the GET operation, with the COPY option, and to check the file status:

1. Open
2. Endfile
3. Invalid

Linkage: As for IHEIOAA

Called by: Compiled code

Entry point IHEIOAC

Function:

To initialize the GET operation with the SKIP option, and to check the file status:

1. Open
2. Endfile
3. Invalid

Linkage:

RA: A(Parameter list)

Parameter list:
A(DCLCB)
A(Abnormal return)
A(Expression value)

Called by: Compiled code

Entry point IHEIOAT

Function:

To terminate the GET operation.

Linkage: None

Called by: Compiled code

IHEIOB

Calls:

IHEIOP, IHEOCL

Entry point IHEIOBA

Function:

To initialize the PUT operation, and to check the file status:

1. Open
2. Transmit error
3. Invalid

Linkage:

RA: A(Parameter list)
Parameter list:
A(DCLCB)
A(Abnormal return)

Called by: Compiled code

Entry point IHEIOBB

Function:

To initialize PUT, and perform PAGE, and to check the file status:

1. Open
2. Transmit error
3. Invalid

Linkage: As for IHEIOBA

Called by: Compiled code

Entry point IHEIOBC

Function:

To initialize PUT, and perform SKIP, and to check the file status:

1. Open
2. Transmit error
3. Invalid

Linkage:

RA: A(Parameter list)
Parameter list:
  A(DCLCB)
  A(Abnormal return)
  A(Expression value)

Called by: Compiled code

Entry point IHEIOBD

Function:

To initialize PUT, and perform LINE, and to check the file status:

1. Open
2. Transmit error
3. Invalid

Linkage: As for IHEIOBC

Called by: Compiled code

Entry point IHEIOBE

Function:

To initialize PUT, and perform PAGE and LINE, and to check the file status:

1. Open
2. Transmit error
3. Invalid

Linkage: As for IHEIOBC

Called by: Compiled code

Entry point IHEIOBT

Function:

To terminate the PUT operation.

Linkage: None

Called by: Compiled code

IHEIOC

Calls: IHESA, IHETSA

Entry point IHEIOCA

Function:

To initialize the GET operation, with the STRING option.

Linkage:

RA: A(Parameter list)
Parameter list:
  A(SDV)
  A(DED)

Called by: Compiled code

Entry point IHEIOCB

Function:

To initialize the GET operation, with the STRING and COPY options.

Linkage: As for IHEIOCA

Called by: Compiled code

Entry point IHEIOCC

Function:

To initialize the PUT operation, with the STRING option.

Linkage: As for IHEIOCA

Called by: Compiled code

Entry point IHEIOCT

Function:

To terminate the GET or PUT operations, with the STRING option.

Linkage: None

Called by: Compiled code

IHEIOD

Calls: IHEIOF, IHESA, IHEPRT, IHEPTT, IHETSA

Entry point IHEIODG

Function:

To obtain the next data field from the record buffer(s).

Linkage:

Library communication area (WSDV)

Called by: Format directors, IHEIOX

Entry point IHEIODP

Function:

To obtain space for a data field in the record buffer(s).

Linkage: As for IHEIODG

Called by: Format directors, IHEIOX

Entry point IHEIODT

Function:

To terminate the data field request.

Linkage: As for IHEIODG

Called by: Format directors

IHEIOF

Calls: Data management (QSAM)

Entry point: IHEIOFA

Function:

To obtain logical records via data management interface modules, and initialize FCB record pointers and counters.

Linkage: RA: A(FCB)

Called by:

IHEDD, IHEDD, IHEDDP, IHEDDT, IHEIOD, IHEIOP, IHEIOX, IHELDI, IHELDO, IHEOCL, IHEOCT, IHEPRT, IHEPTT

IHEIOG

Entry point: IHEIOGA

Function:

As for IHEION

IHEION

This module is used in a non-multitasking environment and is equivalent to module IHEINT in a multitasking environment.

Calls:

Supervisor(FREEMAIN, GETMAIN), IHEITB, IHEITC, IHEITD, IHEITE, IHEITF, IHEITG, IHEOCL

Entry point: IHEIONA

Function:

To verify a RECORD I/O request and to invoke the appropriate data management interface module to perform the required operation, in a non-multitasking environment.

Linkage:

```
RA: A(Parameter list)
Parameter list:
  A(DCLCB)
  A(RDV)/(IGNORE factor)
  A(EVENT variable)/(0)/A(Error return)
  A(KEY|KEYFROM|KEYTO SDV)/(0)
  A(Request control block)
```

Called by: Compiled code

IHEIOP

Calls: IHEIOF

Entry point IHEIOPA

Function: PAGE option/format.

Linkage: No explicit parameters

Called by: Compiled code, IHEIOB

Entry point IHEIOPB

Function: SKIP option/format.

Linkage:

RA: A(FED)
FED: Halfword binary integer

Called by: Compiled code, IHEIOA, IHEIOB

Entry point IHEIOPC

Function: LINE option/format.

Linkage: As for IHEIOPB

Called by: As for IHEIOPA

IHEIOX

Calls: IHEIOD, IHEIOF

Entry point IHEIOXA

Function:

To skip next n characters in record(s).

Linkage:

RA: A(FED)
FED: Halfword binary integer

Called by: Compiled code

Entry point IHEIOXB

Function:

To place n blanks in record(s).

Linkage: As for IHEIOXA

Called by: Compiled code

Entry point IHEIOXC

Function: To position to COLUMN(n).

Linkage: As for IHEIOXA

Called by: Compiled code

IHEITB

Calls:

Data management (BSAM), Supervisor (CHAP, GETMAIN)

Entry point: IHEITBA

Function:

To provide the interface with BSAM for:

1. CONSECUTIVE data sets with the UNBUFFERED attribute.

2. REGIONAL data sets, whether or not UNBUFFERED, opened for INPUT/UPDATE

Linkage:

RA: A(FCB)
RB: A(Parameter list)
Parameter list:
  A(DCLCB)
  A(RDV)/A(IOCB)/A(IGNORE factor)/A(SDV)
  A(Event variable)/(0)
  A(KEY|KEYFROM|KEYTO SDV)/(0)
  A(Request control block)

| Called by: IHEION, IHEINT

IHEITC

Calls:

Data management (BSAM), Supervisor (CHAP, GETMAIN)

Entry point: IHEITCA

Function:

To provide the interface with BSAM for creating REGIONAL data sets when opened for SEQUENTIAL output.

Linkage:

RA: A(FCB)
RB: A(Parameter list)
Parameter list:
  A(DCLCB)
  A(RDV)/A(IOCB)
  A(Event variable)/(0)/A(Abnormal return)
  A(KEY|KEYFROM SDV)/(0)
  A(Request control block)

| Called by: IHEION, IHEINT, IHEOCL

IHEITD

Calls:

Data management (QISAM), Supervisor (GETMAIN), IHESA, IHETSA

Entry point: IHEITDA

Function:

To provide the interface with QISAM for creating or accessing INDEXED data sets when opened for SEQUENTIAL access.

Linkage:

RA: A(FCB)
RB: A(Parameter list)
Parameter list:
  A(DCLCB)
  A(RDV)/A(SDV)
  A(Error return)/(0)
  A(KEY|KEYFROM|KEYTO SDV)/(0)
  A(Request control block)

| Called by: IHEION, IHEINT

IHEITE

Calls:

Data management (BISAM), Supervisor (GETMAIN), IHESA

Entry point: IHEITEA

Function:

To provide the interface with BISAM for accessing INDEXED data sets opened for DIRECT access in a non-multitasking environment.

Linkage:

```
RA: A(FCB)
RB: A(Parameter list)
Parameter list:
  A(DCLCB)
  A(RDV)/A(IOCB)/A(SDV)
  A(Event variable)/(0)
  A(KEY|KEYFROM SDV)/(0)
  A(Request control block)
```

Called by: IHEION

IHEITF

Calls:

Data management (BDAM), Supervisor (GETMAIN), IHESA

Entry point: IHEITFA

Function:

To provide the interface with BDAM for REGIONAL data sets opened for DIRECT access in a non-multitasking environment.

Linkage:

```
RA: A(FCB)
RB: A(Parameter list)
Parameter list:
  A(DCLCB)
  A(RDV)/A(IOCB)/A(SDV)
  A(Event variable)/(0)
  A(KEY|KEYFROM SDV)/(0)
  A(Request control block)
```

Called by: IHEION

IHEITG

Calls: Data management (QSAM)

Entry point: IHEITGA

Function:

To provide the interface with QSAM for CONSECUTIVE data sets opened for RECORD I/O with the BUFFERED attribute.

Linkage:

```
RA: A(FCB)
RB: A(Parameter list)
Parameter list:
  A(DCLCB)
  A(RDV)/A(SDV)
  A(Error return)/(0)
  A(0)
  A(Request control block)
```

Called by: IHEION, IHEINT

IHEITH

Calls:

Data management (BISAM), Supervisor (CHAP, DEQ, ENQ, GETMAIN), IHETSA

Entry point: IHEITHA

Function:

To provide the interface with BISAM for accessing INDEXED data sets opened for DIRECT access in a multitasking environment.

Linkage:

```
RA: A(FCB)
RB: A(Parameter list)
Parameter list:
  A(DCLCB)
  A(RDV)/A(IOCB)/A(SDV)
  A(Event variable) /(0)
  A(KEY | KEYFROM SDV)/(0)
  A(Request control block)
```

Called by: IHEINT

IHEITJ

Calls:

Data management (BDAM), Supervisor (CHAP, DEQ, ENQ, GETMAIN), IHETSA

Entry point: IHEITJA

Function:

To provide the interface with BDAM for REGIONAL data sets opened for DIRECT access in a multitasking environment.

Linkage:

```
RA: A(FCB)
RB: A(Parameter list)
Parameter list:
  A(DCLCB)
  A(RDV)/A(IOCB)/A(SDV)
  A(Event variable)/(0)
  A(KEY | KEYFROM SDV) /(0)
  A(Request control block)
```

Called by: IHEINT

IHEITK

Calls:

Data Management (QSAM), Supervisor (GETMAIN, FREEMAIN)

Entry point: IHEITKA

Function:

To provide the interface with QSAM for consecutive data sets opened for RECORD I/O Input with the BUFFERED attribute and VS or VBS format records.

Linkage:

RA: A(FCB)
RB: A(Parameter list)
Parameter list:
  A(DCLCB)
  A(RDV)/A(SDV)
  A(Error Return)/(0)
  A(0)
  A(Request Control Block)

Called by: IHEION, IHEINT

IHEITL

Calls:

Data Management (QSAM), Supervisor (GETMAIN, FREEMAIN)

Entry point: IHEITLA

Function:

To provide the interface with QSAM for consecutive data sets opened for RECORD I/O Output with the BUFFERED attribute and VS or VBS format records.

Linkage: as for IHEITK

Called by: IHEION, IHEINT, IHEOCL

IHEJXI

Calls: IHESA, IHETSA

Entry point IHEJXII

Function:

To initialize IHEJXI to give bit addresses, and to find the first element of the array.

Linkage:

RA: A(ADV)
RB: A(Number of dimensions)
On return:
RA: Bit address of first element

Called by: IHENL2, IHESTG

Entry point IHEJXIY

Function:

As for IHEJXII but for byte addresses.

Linkage:

RA: A(ADV)
RB: A(Number of dimensions)
On return:
RA: A(First element)

Called by:

IHEOSW, IHEPDF, IHEPDL, IHEPDS, IHEPDW, IHEPDX, IHEPDZ, IHESMF, IHESMG, IHESMH, IHESMX, IHESTG

Entry point IHEJXIA

Function:

To find the next element of the array.

Linkage:

No explicit arguments
Implicit arguments:
  LCA
  VDA, obtained in initialization
On return:
RA: Bit or byte address of the next element
BR=0: Normal return
BR=4: If the address of the last element of the array was provided on the previous normal return

Called by:

All modules calling IHEJXII and IHEJXIY

IHEJXS

Entry point IHEJXSI

Function:

To find the first and last elements of an array and to give their addresses as bit addresses.

Linkage:

RA: A(ADV)
RB: A(Number of dimensions)
On Return:
R0: Bit address of first element
RA: Bit address of last element

Called by: IHENL1

Entry point IHEJXSY

Function:

As for IHEJXSI but for byte addresses.

Linkage:

RA: A(ADV)
RB: A(Number of dimensions)
On return:
R0: A(First element)
RA: A(Last element)

Called by:

IHEPSF, IHEPSL, IHEPSS, IHEPSW, IHEPSX, IHEPSZ, IHESSF, IHESSG, IHESSH, IHESSX

IHEKCA

Entry point: IHEKCAA

Function:

To check that external data with a decimal picture specification is valid for that specification.

Linkage:

RA: A(Source)
RB: A(Source DED)

Called by: IHEDIE, IHEDIM

IHEKCB

Entry point: IHEKCBA

Function:

To check that external data with a sterling picture specification is valid for that specification.

Linkage:

RA: A(Source)
RB: A(Source DED)

Called by: IHEDIE

IHEKCD

Entry point IHEKCDA

Function:

To check that external data with a character picture specification is valid for that specification. The ONSOURCE address is stored.

Linkage:

RA: A(Source)
RB: A(Source DED)

Called by: IHEDIB, IHELDI

Entry point IHEKCDB

Function:

As for IHEKCDA, but the ONSOURCE address is not stored.

Linkage: As for IHEKCDA

Called by: As for IHEKCDA

IHELDI

Calls:

IHEDCN, IHEDMA, IHEDNB, IHEDNC, IHEIOF, IHEKCD, IHEPRT, IHEPTT, IHESA, IHETSA, IHEVCA, IHEVCS, IHEVSC, IHEVSD

Entry point IHELDIA

Function:

To read data from an input stream and to assign it to internal variables according to list-directed input conventions.

Linkage:

RA: A(Parameter list)
Parameter list:
A(Variable$_1$)
A(DED$_1$)
.
.
.
A(Variable$_n$)
A(DED$_n$)
(High-order byte of last argument indicates end of parameter list.)

Called by: Compiled code

Entry point IHELDIB

Function:

As for IHELDIA but for single variables.

Linkage:

RA: A(Variable)
RB: A(DED)

Called by: Compiled code

Entry point IHELDIC

Function:

To scan the value field (entry for data-directed input).

Linkage:

```
RA: A(Buffer SDV)
RB: A(Control block)
Control block: H'VDA count so far'
               X'Flag box'(one byte)
Return codes:
  BR=0: Not last item
  BR=4: Last item
  BR=8: End  of file encountered before
        complete data field collected
```

Called by: IHEDDI

Entry point IHELDID

Function:

To assign a value to a variable (entry for data-directed input).

Linkage:

```
RA: A(Variable)
RB: A(DED)
RC: A(Control block)
Control block: H'VDA count so far'
               X'Flag box' (one byte)
```

Called by: IHEDDI

IHELDO

Calls: IHEDNC, IHEIOF, IHEVSB

Entry point IHELDOA

Function:

To prepare data for output according to list-directed output conventions, and to place it in an output stream.

Linkage:

RA: A(Parameter list)
Parameter list:
  A(Variable$_1$)
  A(DED$_1$)
  .
  .
  .
  A(Variable$_n$)
  A(DED$_n$)
  (High-order byte of last argument indicates end of parameter list.)

Called by: Compiled code

Entry point IHELDOB

Function:

As for IHELDOA, but for only one item of the list of data.

Linkage:

RA: A(Variable)
RB: A(DED)

Called by: Compiled code

Entry point IHELDOC

Function:

As for IHELDOA, but used by data-directed output.

Linkage:

RA: A(Variable)
RB: A(DED)
RC: A(FCB)

Called by: IHEDDO

IHELNL

Entry point IHELNLE

Function:

LOG(x), where x is real long floating-point.

Linkage:

RA: A(Parameter list)
Parameter list:
  A(x)
  A(Target)

Called by:

Compiled code, IHEHTL, IHELNZ, IHEXXL, IHEXXZ

Entry point IHELNL2

Function:

LOG2(x), where x is real long floating-point.

Linkage: As for IHELNLE

Called by: As for IHELNLE

Entry point IHELNLD

Function:

LOG10(x), where x is real long floating-point.

Linkage: As for IHELNLE

Called by: As for IHELNLE

IHELNS

Entry point IHELNSE

Function:

LOG(x), where x is real short floating-point.

Linkage:

```
RA: A(Parameter list)
Parameter list:
  A(x)
  A(Target)
```

Called by:

Compiled code, IHEHTS, IHELNW, IHEXXS, IHEXXW

Entry point IHELNS2

Function:

LOG2(x), where x is real short floating-point.

Linkage: As for IHELNSE

Called by: As for IHELNSE

Entry point IHELNSD

Function:

LOG10(x), where x is real short floating-point.

Linkage: As for IHELNSE

Called by: As for IHELNSE

IHELNW

Calls: IHEATS, IHELNS

Entry point: IHELNW0

Function:

LOG(z), where z is complex short floating-point.

Linkage:

```
RA: A(Parameter list)
Parameter list:
  A(z)
  A(Target)
```

Called by: Compiled code, IHEXXW

IHELNZ

Calls: IHEATL, IHELNL

Entry point: IHELNZ0

Function:

LOG(z), where z is complex long floating-point.

Linkage:

```
RA: A(Parameter list)
Parameter list:
  A(z)
  A(Target)
```

Called by: Compiled code, IHEXXZ

IHELSP

Calls: Supervisor (FREEMAIN,GETMAIN)

Function:

Storage management for list processing.

Entry point IHELSPA

Function:

To provide storage in an area variable for an allocation of a based variable.

Linkage:

```
RA: A(Eight-byte word-aligned parameter
    list)
RB: A(ALLOCATE statement)
Parameter list:
  Byte 0: Not used
  Bytes 1-3: A(Area variable)
  Byte 4: Offset of beginning of based
          variable from doubleword
          boundary
  Bytes 5-7: Length of based variable
```

On return:

```
RA: A(Eight-byte word-aligned parameter
    list)
```

Parameter list:
  Byte 0: Not used
  Bytes 1-3: A(Based variable)
  Byte 4: Offset of beginning of based variable from doubleword boundary
  Bytes 5-7: Length of based variable

Called by: Compiled code

Entry point IHELSPB

Function:

To free storage allocated to a based variable in an area variable.

Linkage:

RA: A(Eight-byte word-aligned parameter list)
RB: A(Area variable)
Parameter list:
  Byte 0: Not used
  Bytes 1-3: A(Based variable)
  Byte 4: Offset of beginning of based variable from doubleword boundary
  Bytes 5-7: Length of based variable

Called by: Compiled code

Entry point IHELSPC

Function:

Assignments between area variables.

Linkage:

RA: A(Source area variable)
RB: A(Target area variable)

Called by: Compiled code.

Entry point IHELSPD

Function:

To provide system storage for an allocation of a based variable (using GETMAIN macro).

Linkage:

RA: A(Eight-byte word-aligned parameter list)

Parameter list:

Bytes 0-3: Not used

Byte 4: Offset of beginning of based variable from doubleword boundary

Bytes 5-7: Length of based variable

On return:

RA: A(Eight-byte word-aligned parameter list)
Parameter list:
  Byte 0: Not used
  Bytes 1-3: A(Based variable)
  Bytes 4-7: Not used

Called by: Compiled code

Entry point IHELSPE

Function:

To free system storage allocated to a based variable (using FREEMAIN macro).

Linkage:

RA: A(Eight-byte word-aligned parameter list)

Parameter list:
  Byte 0: Not used
  Bytes 1 - 3: A(Based variable)
  Byte 4: Offset of beginning of based variable from doubleword boundary
  Bytes 5 - 7: Length of based variable

Called by: Compiled code

IHEM91

Calls: IHEERR

Entry point IHEM91A

Function:

1. To analyze the exception or exceptions in an imprecise interrupt on a Model 91

2. To set up a list of these exceptions (in LWE)

3. To raise the first of a series of PL/I conditions corresponding to these exceptions

Linkage:

PSW at interrupt is in current LWE + 112

Called by:

IHEERR, when an imprecise interrupt is detected

Entry point IHEM91B

Function:

To continue raising, in succession, the

PL/I conditions corresponding to the exceptions

Linkage:

List of exceptions is in current LWE + 136

Called by: IHEERR

Entry point IHEM91C

Function:

To print an error message for each unprocessed exception when, as a result of the processing of an earlier exception in the list, a program is forced to terminate before processing of the list is complete

Linkage: None

Called by: IHEERR

IHEMAI

Entry point: IHEMAIN

Function:

Contains address of IHEBEGN; loaded only if there is no main procedure.

Linkage: None

Called by: IHESA, IHETSA

IHEMPU

Entry point: IHEMPU0

Function:

MULTIPLY(w,z,p,q), where w and z are complex fixed binary, and (p,q) is the target precision.

Linkage:

```
RA: A(Parameter list)
Parameter list:
  A(w)
  A(DED for w)
  A(z)
  A(DED for z)
  A(Target)
  A(DED for target)
```

Called by: Compiled code

IHEMPV

Calls: IHEAPD

Entry point: IHEMPV0

Function:

MULTIPLY(w,z,p,q), where w and z are complex fixed decimal, and (p,q) is the target precision.

Linkage:

```
RA: A(Parameter list)
Parameter list:
  A(w)
  A(DED for w)
  A(z)
  A(DED for z)
  A(Target)
  A(DED for target)
```

Called by: Compiled code

IHEMSI

Entry point: IHEMSIA

Function:

To call IHEERRC so that an error message is printed saying that STIMER facilities are unavailable.

IHEMST

Entry Point: IHEMSTA

Function:

To call IHEERRC so that an error message is printed saying that the TIME facility is unavailable.

Called by: Compiled code

IHEMSW

Calls:

Supervisor (FREEMAIN, WAIT), I/O transmit module whose address is in the FCB.

Entry point: IHEOSWA

Function:

1. According to the count passed, to return to the caller or to wait until a single I/O event is complete. If the count is ≤0, immediate return is made; otherwise the event is waited on.
2. To branch to the I/O transmit module to raise I/O conditions if necessary.

Linkage:

```
RA: A(Parameter list)
Parameter list:
  A(Count)
  A(Event variable)
```

Called by: Compiled code

IHEMXB

Entry point IHEMXBX

Function:

MAX($x_1,x_2,\ldots,x_n$), where x_1,x_2 and x_n are real fixed-point binary.

Linkage:

```
RA: A(Parameter list)
Parameter list:
  A(x1)
  A(DED for x1)
    .
    .
    .
  A(xn)
  A(DED for xn)
  A(Target)
  A(Target DED)
  (High-order byte of last argument
   indicates end of parameter list.)
```

Called by: Compiled code

Entry point IHEMXBN

Function:

MIN($x_1,x_2,\ldots,x_n$), where x_1,x_2 and x_n are real fixed-point binary.

Linkage: As for IHEMXBX

Called by: Compiled code

IHEMXD

Entry point IHEMXDX

Function:

MAX($x_1,x_2,\ldots,x_n$), where x_1,x_2 and x_n are real fixed-point decimal.

Linkage:

```
RA: A(Parameter list)
Parameter list:
  A(x1)
  A(DED for x1)
    .
    .
    .
  A(xn)
  A(DED for xn)
  A(Target)
  A(Target DED)
  (High-order byte of last argument
   indicates end of parameter list.)
```

Called by: Compiled code

Entry point IHEMXDN

Function:

MIN($x_1,x_2\ldots,x_n$), where x_1,x_2 and x_n are real fixed-point decimal.

Linkage: As for IHEMXDX

Called by: Compiled code

IHEMXL

Entry point IHEMXLX

Function:

MAX($x_1,x_2,\ldots,x_n$), where x_1,x_2 and x_n are real long floating-point.

Linkage:

```
RA: A(Parameter list)
Parameter list:
  A(x1)
  A(x2)
    .
    .
    .
  A(xn)
  A(Target)
  (High-order byte of last argument
   indicates end of parameter list.)
```

Called by: Compiled code

Entry point IHEMXLN

Function:

MIN($x_1,x_2,\ldots,x_n$), where x_1,x_2 and x_n are real long floating-point.

Linkage: As for IHEMXLX

Called by: Compiled code

IHEMXS

Entry point IHEMXSX

Function:

MAX($x_1,x_2,\ldots,x_n$), where x_1,x_2 and x_n are real short floating-point.

Linkage:

RA: A(Parameter list)
Parameter list:
A(x_1)
A(x_2)
.
.
.
A(x_n)
A(Target)
(High-order byte of last argument indicates end of parameter list.)

Called by: Compiled code

Entry point IHEMXSN

Function:

MIN($x_1, x_2, \ldots, x_n$), where x_1, x_2 and x_n are real short floating-point.

Linkage: As for IHEMXSX

Called by: Compiled code

IHEMZU

Entry point IHEMZUM

Function:

$z_1 * z_2$, where z_1 and z_2 are complex fixed-point binary.

Linkage:

RA: A(z_1)
*RB: A(DED for z_1)
RC: A(z_2)
*RD: A(DED for z_2)
RE: A(Target)
*RF: A(Target DED

Called by: Compiled code, IHEXIU

Entry point IHEMZUD

Function:

z_1/z_2, where z_1 and z_2 are complex fixed-point binary.

Linkage:

RA: A(z_1)
RB: A(DED for z_1)
RC: A(z_2)
*RD: A(DED for z_2)
RE: A(Target)
*RF: A(Target DED)

Called by: Compiled code

IHEMZV

Entry point IHEMZVM

Function:

$z_1 * z_2$, where z_1 and z_2 are complex fixed-point decimal.

Linkage:

RA: A(z_1)
RB: A(DED for z_1)
RC: A(z_2)
RD: A(DED for z_2)
RE: A(Target)
*RF: A(Target DED)

Called by: Compiled code, IHEXIV

Entry point IHEMZVD

Function:

'z_2. where z_1 and z_2 are complex fixed-point decimal.

Linkage: As for IHEMZVM

Called by: Compiled code

IHEMZW

Entry point: IHEMZW0

Function:

$z_1 * z_2$, where z_1 and z_2 are complex short floating-point.

Linkage:

RA: A(z_1)
RB: A(z_2)
RC: A(Target)

Called by: Compiled code,IHEXIW

IHEMZZ

Entry point: IHEMZZ0

Function:

$z_1 * z_2$, where z_1 and z_2 are complex long floating-point.

Linkage:

RA: A(z_1)
RB: A(z_2)
RC: A(Target)

Called by: Compiled code,IHEXIZ

IHENL1

Calls: IHEBSA, IHEBSF, IHEBSO, IHEJXS

Entry point IHENL1A

Function:

ALL or ANY for a simple array (or an interleaved array of VARYING elements) of byte-aligned elements and a byte-aligned target.

Linkage:

```
RA: A(Parameter list)
Parameter list:
  A(SADV)
  A(Number of dimensions)
  A(DED of the array)
 (A(IHEBSA0) for ALL, or
 (A(IHEBSO0) for ANY
  A(SDV for Target field )
```

Called by: Compiled code

Entry point IHENL1L

Function:

ALL for a simple array (or an interleaved array of VARYING elements) of elements with any alignment, and a target with any alignment.

Linkage:

```
RA: A(Parameter list)
Parameter list:
  A(SADV)
  A(Number of dimensions)
  A(DED of the array)
  A(IHEBSF0)
  A(SDV for target field )
```

Called by: Compiled code

Entry point IHENL1N

Function: As for IHENL1L, but ANY.

Linkage: As for IHENL1L

Called by: Compiled code

IHENL2

Calls: IHEBSA, IHEBSF, IHEBSO, IHEJXI

Entry point IHENL2A

Function:

ALL or ANY for an interleaved array of fixed-length byte-aligned elements and a byte-aligned target.

Linkage:

```
RA: A(Parameter list)
Parameter list:
  A(SADV)
  A(Number of dimensions)
 *A(DED of the array)
 (A(IHEBSA0) for ALL, or
 (A(IHEBSO0) for ANY
  A(SDV for target field)
```

Called by: Compiled code

Entry point IHENL2L

Function:

ALL for an interleaved array of fixed-length elements with any alignment, and a target with any alignment.

Linkage:

```
RA: A(Parameter list)
Parameter list:
  A(SADV)
  A(Number of dimensions)
 *A(DED of the array)
  A(IHEBSF0)
  A(SDV for target field)
```

Called by: Compiled code

Entry point IHENL2N

Function:

ANY for an interleaved array of fixed-length elements with any alignment, and a target with any alignment.

Linkage:

```
RA: A(Parameter list)
Parameter list:
  A(SADV)
  A(Number of dimensions)
 *A(DED of the array)
  A(IHEBSF0)
  A(SDV for target field)
```

Called by: Compiled code

IHEOCL

Calls:

Supervisor (DCBD, FREEMAIN,LINK), IHECLT, IHEIOF, IHEITC, IHEITL, IHEOPN, IHESA

Entry point IHEOCLA

Function:

Explicit open: links to IHEOPNA; handles error conditions detected by IHEOPN, IHEOPO, IHEOPP, IHEOPQ or IHEOPZ.

Linkage:

RA: A(OPEN parameter list)
Parameter list: See IHEOPN

Called by: Compiled code, IHEPRT

Entry point IHEOCLB

Function:

Explicit close: links to IHECLTA.

Linkage:

RA: A(CLOSE parameter list)
Parameter list: See IHECLTA

Called by: Compiled code

Entry point IHEOCLC

Function:

To perform implicit open.

Linkage:

RA: A(OCB)
RB: A(DCLCB)

| Called by: IHEIOA, IHEIOB, IHEION

Entry point IHEOCLD

Function:

Implicit close:

1. When a task is terminated, to close all the files opened in the task (by linking to IHECLTB).

Linkage:

RA: A(PRV of current task)

Called by: IHESA

IHEOCT

Calls:

Supervisor (DCBD, DEQ, FREEMAIN, LINK),
| IHECTT, IHEIOF, IHEITC, IHEITL, IHEOPN, IHETSA

Entry point IHEOCTA

Function:

Explicit open in a multitasking environment: links to IHEOPNA; handles error conditions detected by IHEOPN, IHEOPO, IHEOPP, IHEOPQ or IHEOPZ.

Linkage:

RA: A(OPEN parameter list)
Parameter list: See IHEOPN

Called by: Compiled code, IHEPTT

Entry point IHEOCTB

Function:

Explicit close in a multitasking environment: links to IHECTTA.

Linkage:

RA: A(CLOSE parameter list)
Parameter list: See IHECTTA

Called by: Compiled code

Entry point IHEOCTC

Function:

To perform implicit open in a multitasking environment.

Linkage:

RA: A(OCB)
RB: A(DCLCB)

| Called by: IHEIOA, IHEIBT, IHEINT

Entry point IHEOCTD

Function:

Implicit close:

1. When a task is terminated, to close all the files opened in the task (by linking to IHECTTB).

2. To dequeue all records locked by the task and free the corresponding EXCLUSIVE blocks.

   To set all imcomplete EVENT variables complete, inactive, and abnormal, and to free the associated IOCBs.

Linkage:

RA: A(PRV of current task)

Called by: IHETSA

IHEOPN

Calls:

IHEOPO (via XCTL), IHEOPZ (via LINK), IHESA, IHETSA

Entry point: IHEOPNA

Function:

Open files:
1. Merge declared attributes with OPEN options.
2. Invoke IHEOPO.
3. Invoke IHEOPZ if declared DIRECT OUTPUT (REGIONAL (1), (2) and (3) only).

Linkage:

```
RA: A(Parameter list)
Parameter list:
  A(OPEN Parameter list)
  A(Private Adcons)
OPEN Parameter list:
  A(DCLCB1)
  A(OPEN Control block1)/0
  A(TITLE-SDV1)/0
  (Reserved)
  (Reserved)
  (Reserved)
  A(LINESIZE1)/0
  A(PAGESIZE1)/0
     .
     .
     .
  A(DCLCBn)
  A(OPEN Control blockn)/0
  A(TITLE-SDVn)/0
  (Reserved)
  (Reserved)
  (Reserved)
  A(LINESIZEn)/0
  A(PAGESIZEn)/0
  (High-order byte of last argument
   indicates end of parameter list.)
```

Called by: IHEOCL, IHEOCT

IHEOPO

Calls:

Supervisor (DCB,DCBD,DEVTYPE, GETMAIN), IHEOPP (via XCTL), IHESA, IHETSA

Entry Point: IHEOPOA

Function:

1. To create and format the FCB.

2. To set file register to A(FCB).

Linkage:

```
RA: A(Parameter list)
Parameter list:
  A(IHEOPN Parameter list)
  A(Subparameter list)
Subparameter list:
  XL4'4*n'(where n is the number of files
    to be opened)
  X'Access/Organization Code1'
  AL3(DCLCB1)
  XL4'Merged attribute1'
     .
     .
     .
  X'Access/Organization Coden'
  AL3(DCLCBn)
  XL4'Merged attributen'
```

NOTE: Access/Organization Code is described in the module listing.

Called by: IHEOPN

IHEOPP

Calls:

Supervisor (DCBD, GETMAIN, GETPOOL, OPEN), IHEOPQ (via XCTL), IHESA, IHETSA

Entry point: IHEOPPA

Function:

1. To invoke data management (OPEN macro).

2. To establish defaults at DCB exit.

3. To acquire initial IOCBs for BSAM.

Linkage:

```
RA: A(Parameter list)
Parameter list:
  A(IHEOPN Parameter list)
  A(Subparameter list)
Subparameter list:
  XL4'4*n'(where n is the number of files
  to be opened)
  X'Access/Organization Code1'
  AL3(DCLCB1)
  XL4'Merged attribute1'
     .
     .
     .
  X'Access/Organization Coden'
  AL3(DCLCBn)
  XL4'Merged attributen'
```

NOTE: Access/Organization Code is described in the module listing.

Called by: IHEOPO

IHEOPQ

Calls:

Supervisor (DCBD,FREEPOOL,GETMAIN,LOAD), IHESA, IHETSA

Entry point: IHEOPQA

Function:

1. To load record-oriented I/O interface modules.

2. To link FCBs through the IHEQFOP chain.

3. To acquire the initial IOCBs for BDAM and BISAM linkage.

4. To simulate PUT PAGE when opening a PRINT file.

Linkage:

```
RA: A(Parameter list)
Parameter list:
  A(IHEOPN parameter list)
  A(Subparameter list)
  A(Data management OPEN parameter
    list)

Subparameter list:
  XL4'4*n'  (where  n  is the number of
            files to be opened)
  X'Access/Organization Code_n'
  AL3(DCLCB_1)
  XL4'Merged attributes_1'
      .
      .
      .
  X'Access/Organization Code_n'
  AL3(DCLCB_n)
  XL4'Merged attributes_n'

Data management OPEN parameter list:
  XL4'4*n' (where n is  the  number  of
            files to be opened)
  X(Flags for data management OPEN
     executor_1)
  AL3(DCB_1)
      .
      .
      .
  X(Flags for data management OPEN
     executor_n)
  AL3(DCB_n)
```

NOTE: Access/Organization Code is described in the module listing.

Called by: IHEOPP

IHEOPZ

Calls:

Supervisor (CHECK,CLOSE,DCB, DCBD, FREEMAIN,FREEPOOL,GETBUF,GETMAIN,OPEN)

Entry point: IHEOPZA

Function:

To provide the format for the initial allocation of a volume assigned to a REGIONAL data set when opened for DIRECT OUTPUT.

Linkage:

```
RA: A(Parameter list)
Parameter list:
  A(Merged attributes)
  A(Entry in IHEOPN Parameter list)
  A(DCLCB)
```

Called by: IHEOPN

IHEOSD

Calls: TIME macro

Entry point: IHEOSDA

Function: To obtain current date.

Linkage:

```
RA: A(Parameter list)
Parameter list: A(Target SDV)
```

Called by: Compiled code

IHEOSE

Calls: IHESA, IHETSA(to terminate the task)

Entry point: IHEOSEA

Function:

To terminate the current task abnormally, raising the FINISH condition if it is the major task.

Called by: Compiled code

IHEOSI

Calls: STIMER macro

Entry point: IHEOSIA

Function:

To use the STIMER macro with the WAIT option for the implementation of DELAY.

Linkage:

RA: A(Parameter list)
Parameter list:
Interval of delay, in milliseconds, in a fullword

Called by: Compiled code

IHEOSS

| Calls: IHESA, IHETSA(to terminate the task)

Entry point: IHEOSSA

Function:

To raise the FINISH condition and abnormally terminate the job step.

Linkage: None

Called by: Compiled code

IHEOST

Entry Point: IHEOSTA

Function:

To use the TIME macro to obtain the time of day.

Linkage:

RA: A(Parameter list)
Parameter list: A(Target SDV)

Called by: Compiled code

IHEOSW

Calls:

Supervisor (FREEMAIN,WAIT), IHEJXI, IHESA, I/O transmit module whose address is in the FCB

Entry point: IHEOSWA

Function:

To determine whether a specified number of events has occurred. If not, to wait until the required number is complete, and, in the case of I/O events, to branch to the I/O transmit module (which raises I/O conditions if necessary).

This module is used in a non-multitasking environment.

Linkage:

RA: A(Parameter list)
Parameter list:

Word 1:

1. If all events are to be waited on:
   Byte 0 = X'FF'
   Bytes 1 - 3 not used

2. If a specified number (N) of events is to be waited on:
   Byte 0 = X'00'
   Bytes 1 - 3 = A(N)

Subsequent words (one for each element or array event):

1. Array event:
   Byte 0 = dimensionality
   Bytes 1 - 3 = A(ADV)

2. Element event:
   Byte 0 = X'00'
   Bytes 1 - 3 = A(Event variable)

(High-order byte of last argument indicates end of parameter list.)

Called by: Compiled code

IHEPDF

Calls: IHEDMA, IHEJXI

Entry point: IHEPDF0

Function:

PROD for an interleaved array of real fixed-point binary or decimal elements. Result is real short or long floating-point.

Linkage:

RA: A(Parameter list)
Parameter list:
A(ADV)
A(Number of dimensions)
A(DED of the array)
A(Target)
A(DED for target)

Called by: Compiled code

IHEPDL

Calls: IHEJXI

Entry point: IHEPDL0

Function:

PROD for an interleaved array of real long floating-point elements. Result is real long floating-point.

Linkage:

RA: A(Parameter list)
Parameter list:
  A(ADV)
  A(Number of dimensions)
  A(Target)

Called by: Compiled code

<u>IHEPDS</u>

Calls: IHEJXI

Entry point: IHEPDS0

Function:

PROD for an interleaved array of real short floating-point elements. Result is real short floating-point.

Linkage:

RA: A(Parameter list)
Parameter list:
  A(ADV)
  A(Number of dimensions)
  A(Target)

Called by: Compiled code

<u>IHEPDW</u>

Calls: IHEJXI

Entry point: IHEPDW0

Function:

PROD for an interleaved array of complex short floating-point elements. Result is complex short floating-point.

Linkage:

RA: A(Parameter list)
Parameter list:
  A(ADV)
  A(Number of dimensions)
  A(Target)

Called by: Compiled code

<u>IHEPDX</u>

Calls: IHEDMA, IHEJXI

Entry point: IHEPDX0

Function:

PROD for an interleaved array of complex fixed-point binary or decimal elements. Result is complex short or long floating-point.

Linkage:

RA: A(Parameter list)
Parameter list:
  A(ADV)
  A(Number of dimensions)
  A(DED of the array)
  A(Target)
  A(DED for target)

Called by: Compiled code

<u>IHEPDZ</u>

Calls: IHEJXI

Entry point: IHEPDZ0

Function:

PROD for an interleaved array of complex long floating-point elements. Result is complex long floating-point.

Linkage:

RA: A(Parameter list)
Parameter list:
  A(ADV)
  A(Number of dimensions)
  A(Target)

Called by: Compiled code

<u>IHEPRT</u>

Calls:

Supervisor (WTO, EXTRACT), IHEIOF,
| IHEOCL, IHESA

<u>Entry point IHEPRTA</u>

Function:

To COPY a data field on the SYSPRINT file, opening it if necessary.

Linkage:

RA: A(Character string)
RB: A(Halfword containing length of character string)

Called by: IHEIOD,IHELDI

Entry point IHEPRTB

Function:

To write an error message on the SYSPRINT file, opening it if necessary. Also, to prepare for system action for CHECK condition.

Linkage: As for IHEPRTA

Called by: IHEDDO, IHEERR, IHEESM, IHEESS

IHEPSF

Calls: IHEDMA, IHEJXS

Entry point: IHEPSF0

Function:

PROD for a simple array of real fixed-point binary or decimal elements. Result is real short or long floating-point.

Linkage:

```
RA: A(Parameter list)
Parameter list:
  A(ADV)
  A(Number of dimensions)
  A(DED of the array)
  A(Target)
  A(DED for target)
```

Called by: Compiled code

IHEPSL

Calls: IHEJXS

Entry point: IHEPSL0

Function:

PROD for a simple array of real long floating-point elements. Result is real long floating-point.

Linkage:

```
RA: A(Parameter list)
Parameter list:
  A(ADV)
  A(Number of dimensions)
  A(Target)
```

Called by: Compiled code

IHEPSS

Calls: IHEJXS

Entry point: IHEPSS0

Function:

PROD for a simple array of real short floating-point elements. Result is real short floating-point.

Linkage:

```
RA: A(Parameter list)
Parameter list:
  A(ADV)
  A(Number of dimensions)
  A(Target)
```

Called by: Compiled code

IHEPSW

Calls: IHEJXS

Entry point: IHEPSW0

Function:

PROD for a simple array of complex short floating-point elements. Result is complex short floating-point.

Linkage:

```
RA: A(Parameter list)
Parameter list:
  A(ADV)
  A(Number of dimensions)
  A(Target)
```

Called by: Compiled code

IHEPSX

Calls: IHEDMA, IHEJXS

Entry point: IHEPSX0

Function:

PROD for a simple array of complex fixed-point binary or decimal elements. Result is complex short or long floating-point.

Linkage:

```
RA: A(Parameter list)
Parameter list:
  A(ADV)
  A(Number of dimensions)
  A(DED of the array elements)
  A(Target)
  A(DED for target)
```

Called by: Compiled code

IHEPSZ

Calls: IHEJXS

Entry point: IHEPSZ0

Function:

PROD for a simple array of complex long floating-point elements. Result is complex long floating-point.

Linkage:

RA: A(Parameter list)
Parameter list:
  A(ADV)
  A(Number of dimensions)
  A(Target)

Called by: Compiled code

IHEPTT

This module is used in a multitasking environment and is equivalent to module IHEPRT in a non-multitasking environment.

Calls:

Supervisor (DEQ, ENQ, EXTRACT, WTO), IHEIOF, IHEOCT, IHETSA

Entry point IHEPTTA

Function:

To COPY a data field on the SYSPRINT file, opening it if necessary, in a multitasking environment.

Linkage:

RA: A(Character string)
RB: A(Halfword containing length of character string)

Called by: IHEIOD, IHELDI

Entry point IHEPTTB

Function:

To write, in a multitasking environment, an error message on the SYSPRINT file, opening it if necessary. Also, to prepare for system action for CHECK condition.

Linkage: As for IHEPTTA

Called by: IHEDDT, IHEERR, IHEESM, IHEESS

IHESA

Calls:

Supervisor (FREEMAIN, GETMAIN, SPIE), IHEBEG, IHEMAI, IHEOCL

Function:

Storage management in a non-multitasking environment.

Entry point IHESADA (Get DSA):

Function:

To provide a DSA for a procedure or begin block and to set DR to point to it.

Linkage:

R0: Length of DSA
DR: A(Current save area)

Called by: Prologues

Entry point IHESADB (Get VDA):

Function:

To get a VDA for compiled code; sets RA=A(VDA).

Linkage:

R0: Length of VDA (excluding control words)
DR: A(Current save area)

Called by: Compiled code

Entry point IHESADD (Get CONTROLLED variable):

Function:

To provide storage for an allocation of a controlled variable, and to place the address of its fourth word in its pseudo-register.

Linkage:

R0: Length of area (not including control words)
RA: A(Controlled-variable pseudo-register)

Called by: Compiled code

Entry point IHESADE (Get LWS):

Function:

To provide a new LWS, and to update the LWS pseudo-registers.

Linkage: None

Called by: Library modules

Entry point IHESADF (Get Library VDA):

Function:

To provide a VDA for library modules and to set RA = A(VDA).

Linkage:

R0: Length of VDA (including control words)

Called by: Library modules

Entry point IHESAFA (END):

Function:

Frees the DSA current at entry together with its associated VDAs. Request to free the DSA of the main procedure results in raising FINISH, closing all opened files, releasing automatic storage to the supervisor and finally returning to the supervisor with a return code of zero.

Linkage: None

Called by: Epilogues

Entry point IHESAFB (RETURN):

Function:

Frees all chain elements up to and including the last procedure DSA in the chain. Can terminate a main procedure as in IHESAFA.

Linkage: None

Called by: Compiled code

Entry point IHESAFC (GO TO):

Function:

The DSA indicated by the invocation count, or pointed to by DR, is made current. All chain elements up to this DSA, with the exception of its VDAs and itself, are freed.

Linkage:

```
RA: A(Eight-byte word-aligned parameter
    list)
Parameter list:
  Word 1 = Either   Invocation   count
           (sign bit of word 2 = 0)
           Or PR offset  (sign  bit  of
           word 2 = 1)
  Word  2 = A(Location to which control
           is to be returned)
```

Called by: Compiled code

Entry point IHESAFD (Free VDA/LWS)

Function:

Frees the VDA or LWS at the end of the DSA chain.

Linkage:

IHEQSLA: A(VDA or LWS to be freed)
(A VDA or LWS can be freed only when it is the last allocation)

Called by: Compiled code, library modules

Entry point IHESAFF (Free controlled variable):

Function:

Frees the latest allocation of a controlled variable, and updates the associated pseudo-register.

Linkage:

RA: A(Controlled variable pseudo-register)

Called by: Compiled code

Entry point IHESAFQ

Function:

To close all files and to return to the supervisor.

Linkage: None

Called by: Library modules

Entry point IHESAPA

Function:

1. To provide a PRV and LWS for a main procedure, and to issue a SPIE macro; then to transfer control to an address constant named IHEMAIN.

2. To pass a PARM parameter from the EXEC card.

Linkage:

L(PRV) from linkage editor
L(LWS) from assembly of IHELIB

Called by: Initial entry

Entry Point IHESAPB

Function:

As for IHESAPA, except that the code handling PARM parameter is bypassed.

Linkage:

L(PRV) from linkage editor
L(LWS) from assembly of IHELIB

Entry point IHESAPC

Function:

As for IHESAPA, but also reserves a 512-byte area for optimization purposes.

Linkage:

L(PRV) from linkage editor
L(LWS) from assembly of IHELIB

Entry point IHESAPD

Function:

As for IHESAPB, but also reserves a 512-byte area for optimization purposes.

Linkage:

L(PRV) from linkage editor
L(LWS) from assembly or IHELIB

Entry point IHESARA

Function:

To restore the environment of a program to what it was before:

1. the execution of an ON statement associated with the on-unit to be entered, or

2. the passing of the entry parameter associated with the called procedure.

Then to branch to the on-unit or the procedure.

Linkage:

RA: A(Parameter list)
Parameter list:
  A(Entry parameter). The entry parameter is an 8-byte field containing:

  1st word: On-unit or entry address

  2nd word: Invocation count of the DSA associated with either the passing procedure or the procedure in which the ON statement was executed

Called by: Compiled code, IHEERR

Entry point IHESARC

Function:

To place the return code in the pseudo-register IHEQRTC.

Linkage:

RA: A(Parameter list)
Parameter list:
  A(Return code) (The return code is fixed binary with default prceision.)

Called by: Compiled code

IHESHL

Calls: IHEEXL

Entry point IHESHLS

Function:

SINH(x), where x is real long floating-point.

Linkage:

RA: A Parameter list)
Parameter list:
  A(x)
  A(Target)

Called by: Compiled code

Entry point IHESHLC

Function:

COSH(x), where x is real long floating-point.

Linkage: As for IHESHLS

Called by: Compiled code

IHESHS

Calls: IHEEXS

Entry point IHESHSS

Function:

SINH(x), where x is real short floating-point.

Linkage:

RA: A(Parameter list)
Parameter list:
  A(x)
  A(Target)

Called by: Compiled code

Entry point IHESHSC

Function:

COSH(x), where x is real short floating-point.

Linkage: As for IHESHSS

Called by: Compiled code

IHESMF

Calls: IHEDMA, IHEJXI

Entry point: IHESMF0

Function:

SUM for an interleaved array of real fixed-point binary or decimal elements. Result is real short or long floating-point.

Linkage:

```
RA: A(Parameter list)
Parameter list:
  A(ADV)
  A(Number of dimensions)
  A(DED of the array)
  A(Target)
  A(DED for target)
```

Called by: Compiled code

IHESMG

Calls: IHEJXI

Entry point IHESMGR

Function:

SUM for an interleaved array of real short floating-point elements. Result is real short floating-point.

Linkage:

```
RA: A(Parameter list)
Parameter list:
  A(ADV)
  A(Number of dimensions)
  A(Target)
```

Called by: Compiled code

Entry point IHESMGC

Function:

SUM for an interleaved array of complex short floating-point elements. Result is complex short floating-point.

Linkage: As for IHESMGR

Called by: Compiled code

IHESMH

Calls: IHEJXI

Entry point IHESMHR

Function:

SUM for an interleaved array of real long floating-point elements. Result is real long floating-point.

Linkage:

```
RA: A(Parameter list)
Parameter list:
  A(ADV)
  A(Number of dimensions)
  A(Target)
```

Called by: Compiled code

Entry point IHESMHC

Function:

SUM for an interleaved array of complex long floating-point elements. Result is complex long floating-point.

Linkage: As for IHESMHR

Called by: Compiled code

IHESMX

Calls: IHEDMA, IHEJXI

Entry point: IHESMX0

Function:

SUM for an interleaved array of complex fixed-point binary or decimal elements. Result is complex short or long floating-point.

Linkage:

```
RA: A(Parameter list)
Parameter list:
  A(ADV)
  A(Number of dimensions)
  A(DED of the array)
  A(Target)
  A(DED for target)
```

Called by: Compiled code

IHESNL

Entry point IHESNLS

Function:

SIN(x), where x is real long floating-point.

Linkage:

RA: A(Parameter list)
Parameter list:

A(x)
A(Target)

Called by: Compiled code, IHEEXZ, IHESNZ

Entry point IHESNLZ

Function:

SIND(x), where x is real long floating-point.

Linkage: As for IHESNLS

Called by: Compiled code

Entry point IHESNLC

Function:

COS(x), where x is real long floating-point.

Linkage: As for IHESNLS

Called by: Compiled code, IHEEXZ, IHESNZ

Entry point IHESNLK

Function:

COSD(x), where x is real long floating-point.

Linkage: As for IHESNLS

Called by: Compiled code

IHESNS

Entry point IHESNSS

Function:

SIN(x), where x is real short floating-point.

Linkage:

RA: A(Parameter list)
Parameter list:
A(x)
A(Target)

Called by: Compiled code, IHEEXW, IHESNW

Entry point IHESNSZ

Function:

SIND(x), where x is real short floating-point.

Linkage: As for IHESNSS

Called by: Compiled code

Entry point IHESNSC

Function:

COS(x), where x is real short floating-point.

Linkage: As for IHESNSS

Called by: Compiled code, IHEEXW, IHESNW

Entry point IHESNSK

Function:

COSD(x), where x is real short floating-point.

Linkage: As for IHESNSS

Called by: Compiled code

IHESNW

Calls: IHEEXS, IHESNS

Entry point IHESNWS

Function:

SIN(z), where z is complex short floating-point.

Linkage:

RA: A(Parameter list)
Parameter list:
A(z)
A(Target)

Called by: Compiled code

Entry point IHESNWZ

Function:

SINH(z), where z is complex short floating-point.

Linkage: As for IHESNWS

Called by: Compiled code

Entry point IHESNWC

Function:

COS(z), where z is complex short floating-point.

Linkage: As for IHESNWS

Called by: Compiled code

Entry point IHESNWK

Function:

COSH(z), where z is complex short floating-point.

Linkage: As for IHESNWS

Called by: Compiled code

IHESNZ

Calls: IHEEXL, IHESNL

Entry point IHESNZS

Function:

SIN(z), where z is complex long floating-point.

Linkage:

RA: A(Parameter list)
Parameter list:
  A(z)
  A(Target)

Called by: Compiled code

Entry point IHESNZZ

Function:

SINH(z), where z is complex long floating-point.

Linkage: As for IHESNZS

Called by: Compiled code

Entry point IHESNZC

Function:

COS(z), where z is complex long floating-point.

Linkage: As for IHESNZS

Called by: Compiled code

Entry point IHESNZK

Function:

COSH(z), where z is complex long floating-point.

Linkage: As for IHESNZS

Called by: Compiled code

IHESQL

Entry point: IHESQL0

Function:

SQRT(x), where x is real long floating-point.

Linkage:

RA: A(Parameter list)
Parameter list:
  A(x)
  A(Target)

Called by: Compiled code, IHEABZ, IHESQZ

IHESQS

Entry point: IHESQS0

Function:

SQRT(x), where x is real short floating-point.

Linkage:

RA: A(Parameter list)
Parameter list:
  A(x)
  A(Target)

Called by: Compiled code, IHEABW, IHESQW

IHESQW

Calls: IHESQS, IHEABW

Entry point: IHESQW0

Function:

SQRT(z), where z is complex short floating-point.

Linkage:

RA: A(Parameter list)
Parameter list:
  A(z)
  A(Target)

Called by: Compiled code

IHESQZ

Calls: IHEABZ, IHESQL

Entry point: IHESQZ0

Function:

SQRT(z), where z is complex long floating-point.

Linkage:

RA: A(Parameter list)
Parameter list:
A(z)
A(Target)

Called by: Compiled code

IHESRC

Entry point IHESRCA

Function:

Returns SDV of erroneous field (ONSOURCE pseudo-variable). If used out of context, the ERROR condition is raised.

Linkage:

RA: A(Parameter list)
Parameter list: A(Dummy SDV)

Entry point IHESRCB

Function:

Assigns erroneous character to target (ONCHAR built-in function). If used out of context, then 'blank' is returned.

Linkage:

RA: A(Parameter list)
Parameter list: A(Target SDV)

Entry point IHESRCC

Function:

Returns SDV of erroneous field (DATAFIELD). If used out of context, a null string is returned.

Linkage: As for IHESRCA

Entry point IHESRCD

Function:

Returns SDV of erroneous character. (ONCHAR pseudo-variable). If used out of context, the ERROR condition is raised.

Linkage: As for IHESRCA

Entry point IHESRCE

Function:

Returns SDV of the name of the file (ONFILE) which caused entry to the current ON block. If used out of context a null string is returned.

Linkage: As for IHESRCA

Entry point IHESRCF

Function:

Returns SDV of erroneous field (ONSOURCE built-in function). If used out of context, a null string is returned.

Linkage: As for IHESRCA

IHESRD

Entry point: IHESRDA

Function:

Returns SDV of current key (ONKEY built-in function). If used out of context, a null string is returned.

Linkage:

RA: A(Parameter list)
Parameter list: A(Dummy SDV)

IHESRT

Calls:

IHESA, IHETSA, Supervisor (GETMAIN, FREEMAIN, LINK, SPIE), SORT

Function:

To call dynamically, through the use of a LINK macro, the operating system SORT/MERGE from within a PL/I procedure, and, optionally, permitting the use of SORT/MERGE user exits E15 and E35 to invoke PL/I exit procedures contained within the calling PL/I procedure.

Entry point IHESRTA

Function:

To call operating system SORT/MERGE to sort a predefined file (SORTIN) placing the sorted records on another predefined file (SORTOUT).

Linkage:

RA: A(Parameter list)
Parameter list:

1. A(A character string which represents the SORT/MERGE control card to describe the sort fields contained in the record.)

2. A(A character string which represents the SORT/MERGE control card to describe the record format of the records which are to be sorted.)

3. A(A fixed binary value specifying the amount of core storage available to SORT/MERGE.)

4. A(A fixed binary value to be used as a return code from the sort. A return code of 0 indicates the successful completion of the sort, 16 indicates an unsuccessful sort operation.)

5. A(SDV for the DD name replacement string). This is an optional parameter.

Called by: Compiled code (PL/I source statement)

Entry point IHESRTB

Function:

To call operating system SORT/MERGE to sort individual records, passed to SORT/MERGE through user exit E15 by a PL/I exit procedure, onto a predefined file (SORTOUT).

Linkage:

RA: A(Parameter list)
Parameter list:
1, 2, 3, and 4 are as for IHESRTA

5. A(The PL/I functional procedure entry name invoked by SORT/MERGE user exit E15. This exit procedure returns a character string representing a record which is to be included in the sort.)

6. as for 5 in IHESRTA

Called by: Compiled code (PL/I source statement)

Entry point IHESRTC

Function:

To call operating system SORT/MERGE to sort a predefined file (SORTIN), passing individual sorted records through SORT/MERGE user exit E35 to a PL/I exit procedure.

Linkage:

RA: A(Parameter list)
Parameter list:
1, 2, 3, and 4 are as for IHESRTA

5. Not used

6. A(The PL/I procedure entry name invoked by SORT/MERGE user exit E35. This exit procedure receives a sorted record from the sort.)

7. as for 5 in IHESRTA

Called by: Compiled code (PL/I source statement)

Entry point IHESRTD

Function:

To call operating system SORT/MERGE to sort individual records passed to the sort by an exit procedure, through user exit E15, and to pass the sorted records, through user exit E35, to an exit procedure.

Linkage:

RA: A(Parameter list)
Parameter list:
  1, 2, 3, and 4 as for IHESRTA
  5. as for IHESRTB
  6. as for IHESRTC
  7. as for 5 in IHESRTA

Called by: Compiled code (PL/I source statement)

<u>IHESSF</u>

Calls: IHEDMA, IHEJXS

Entry point: IHESSF0

Function:

SUM for a simple array of real fixed-point binary or decimal elements. Result is real short or long floating-point.

Linkage:

RA: A(Parameter list)
Parameter list:
  A(ADV)
  A(Number of dimensions)
  A(DED of the array)
  A(Target)
  A(DED for target)

Called by: Compiled code

<u>IHESSG</u>

Calls: IHEJXS

<u>Entry point IHESSGR</u>

Function:

SUM for a simple array of real short floating-point elements. Result is real short floating-point.

Linkage:

RA: A(Parameter list)
Parameter list:
  A(ADV)
  A(Number of dimensions)
  A(Target)

Called by: Compiled code

<u>Entry point IHESSGC</u>

Function:

SUM for a simple array of complex short floating-point elements. Result is complex short floating-point.

Linkage: As for IHESSGR

Called by: Compiled code

<u>IHESSH</u>

Calls: IHEJXS

<u>Entry point IHESSHR</u>

Function:

SUM for a simple array of real long floating-point elements. Result is real long floating-point.

Linkage:

RA: A(Parameter list)
Parameter list:
  A(ADV)
  A(Number of dimensions)
  A(Target)

Called by: Compiled code

<u>Entry point IHESSHC</u>

Function:

SUM for a simple array of complex long floating-point elements. Result is complex long floating-point.

Linkage: As for IHESSHR

Called by: Compiled code

<u>IHESSX</u>

Calls: IHEDMA, IHEJXS

Entry point: IHESSX0

Function:

SUM for a simple array of complex fixed-point binary or decimal elements. Result is complex short or long floating-point.

Linkage:

RA: A(Parameter list)
Parameter list:
  A(ADV)
  A(Number of dimensions)
  A(DED of the array)
  A(Target)
  A(DED for target)

Called by: Compiled code

IHESTG

Calls: IHEJXI

Entry point IHESTGA

Function:

Given a structure dope vector and its DVD, returns a fullword containing the string length which would result from the concatenation of all the elements of the structure.

Linkage:

RA: A(Structure dope vector)
RB: A(DVD)
RC: A(One-word target field)

Called by: Compiled code

Entry point IHESTGB

Function:

Given a structure dope vector and its DVD, assigns the result of concatenating all the elements of the structure to a string target.

Linkage:

RA: A(Structure dope vector)
RB: A(DVD)
RC: A(Target)

Called by: Compiled code

IHESTR

Calls: IHESA, IHETSA

Entry point IHESTRA

Function:

To compute the address of the first element of a structure and the total length of the structure, using a complete structure dope vector. The result in the two-word target field is:

1st word:	A(Start of structure), in bytes and bit offset
2nd word:	Length of structure, in bytes

Linkage:

RA: A(Structure dope vector)
RB: A(DVD)
RC: A(Two-word target)

Called by: Compiled code

Entry point IHESTRB

Function:

Given a partially completed structure dope vector, to map a structure completely, namely:

1. Locating each structure base element on the alignment boundary required by its data type.
2. Calculating the offset of the start of each base element from the byte address of the beginning of the structure.
3. Calculating the multipliers of all arrays appearing in the structure and calculating the offset of the virtual origin of each array from the byte address of the beginning of the structure.
4. Calculating the total length of the structure.
5. Calculating the offset from the maximum alignment boundary in the structure to the byte address of the start of the structure.

The result is a completed structure dope vector, and a target field which contains:

```
0           7 8                          31
┌────────────────────────────────────────┐
│                  Zero                  │
├────────────┬───────────────────────────┤
│   Offset   │          Length           │
└────────────┴───────────────────────────┘
```

Offset:	Offset in bytes from the maximum alignment boundary in the structure to the start of the structure
Length:	Length of structure, in bytes

Linkage: As for IHESTRA

Called by: Compiled code

Entry point IHESTRC

Function:

As for IHESTRB, but using the COBOL structure mapping algorithm.

Linkage: As for IHESTRA

Called by: Compiled code

IHETAB

Base address of table: IHETABS

Function:

This module is a table of default information provided for use at installation or when individual program replacements are required. It contains:

1. Default PAGESIZE, LINESIZE, and left and right margin positions for all PRINT files.

2. Default tabulation positions for list- and data-directed PRINT file output.

IHETCV

Calls: Supervisor (FREEMAIN,GETMAIN)

Entry point IHETCVA

Function:

To provide storage for an allocation of a controlled variable in a multitasking environment, and to place the address of its fourth word in its pseudo-register.

Linkage:

R0: Length of area (excluding control words)
RA: A(Controlled-variable pseudo-register)

Called by: Compiled code

Entry point IHETCVB

Function:

Frees the latest allocation of a controlled variable in the current task, and updates the associated pseudo-register.

Linkage:
RA: A(Controlled-variable pseudo-register)

Called by: Compiled code

IHETEA

| Calls: Supervisor (CHAP,POST,WAIT)

Entry point: IHETEAA

Function: Event variable assignment.

Linkage:

RA: A(Source event variable)

RB: A(Target event variable)

Called by: Compiled code

IHETER

Entry point: IHETERA

Function:

To search for a matching ON field in a multitasking environment by chaining through DSAs and PRV VDAs. A return code is set in register BR to indicate the result of the search.

Linkage: DR: A(LWE)

Called by: IHEERR

IHETEV

| Calls: Supervisor (CHAP,POST,WAIT)

Entry point: IHETEVA

Function:

COMPLETION pseudo-variable (COMPLETION(v) = expression): sets the specified event variable complete or incomplete according to the evaluation of the expression.

Linkage:

RA: A(Parameter list)
Parameter list:
A(Event variable)
A(Fullword to hold completion value (in bit 24))

Called by: Compiled code

IHETEX

Calls:

IHEERT, IHEPTT Supervisor (WTO, LOAD, DELETE, EXTRACT, ENQ, DEQ, PUT)

Entry point IHETEXA

Function:

To generate a message when a task has been terminated while still active due to the freeing of the block in which the task was attached.

Linkage:

RA contains the address of a VDA which

contains space for the creation of the message and the following parameters:

A(IHEPTTB)
A(Symbol table entry for which the task has been terminated)
A(IHEQSPR)

Called by: IHETSA

Entry point IHETEXB

Function:

To generate a message when a task has been abnormally terminated by the operating system.

Linkage:

DR points to an area of storage contaning a save area, an area for the creation of the message and the following parameters:

Completion code
A(Symbol table entry for the task which has been terminated)
A(IHEQSPR)

Called by: IHETSA

IHETHL

Calls: IHEEXL

Entry point: IHETHL0

Function:

TANH(x), where x is real long floating-point.

Linkage:

RA: A(Parameter list)
Parameter list:
A(x)
A(Target)

Called by: Compiled code, IHETNZ

IHETHS

Calls: IHEEXS

Entry point: IHETHS0

Function:

TANH(x), where x is real short floating-point.

Linkage:

RA: A(Parameter list)
Parameter list:
A(x)
A(Target)

Called by: Compiled code, IHETNW

IHETNL

Entry point IHETNLR

Function:

TAN(x), where x is real long floating-point.

Linkage:

RA: A(Parameter list)
Parameter list:
A(x)
A(Target)

Called by: Compiled code, IHETNZ

Entry point IHETNLD

Function:

TAND(x), where x is real long floating-point.

Linkage: As for IHETNLR

Called by: Compiled code

IHETNS

Entry point IHETNSR

Function:

TAN(x), where x is real short floating-point.

Linkage:

RA: A(Parameter list)
Parameter list:
A(x)
A(Target)

Called by: Compiled code, IHETNW

Entry point IHETNSD

Function:

TAND(x), where x is real short floating-point.

Linkage: As for IHETNSR

Called by: Compiled code

IHETNW

Calls: IHETHS, IHETNS

Entry point IHETNWN

Function:

TAN(z), where z is complex short floating-point.

Linkage:

RA: A(Parameter list)
Parameter list:
A(z)
A(Target)

Called by: Compiled code

Entry point IHETNWH

Function:

TANH(z), where z is complex short floating-point.

Linkage: As for IHETNWN

Called by: Compiled code

IHETNZ

Calls: IHETHL, IHETNL

Entry point IHETNZN

Function:

TAN(z), where z is complex long floating-point.

Linkage:

RA: A(Parameter list)
Parameter list:
A(z)
A(Target)

Called by: Compiled code

Entry point IHETNZH

Function:

TANH(z), where z is complex long floating-point.

Linkage: As for IHETNZN

Called by: Compiled code

IHETOM

Calls: Supervisor (WTO, EXTRACT)

Entry point IHETOMA

Function:

Issues WTO macro instruction if the program does not have a main procedure.

Linkage:

DR points to an area of storage which is used as a save area and as workspace to build up the message.

Called by: IHEBEG

Entry point IHETOMB

Function:

Issues WTO macro instruction if the PRV is longer than 4096 bytes.

Linkage:

As for IHETOMA

Called by: IHEBEG

Entry point IHETOMC

Function:

Issues WTO macro instruction if there has been an interrupt in the error handler.

Linkage:

As for IHETOMA

Called by: IHEERR

Entry point IHETOMD

Function:

Issues WTO macro instruction if the major task of a multitasking program has been terminated with an ABEND. The message contains the completion code.

Linkage:

As for IHETOMA but in addition the completion code is passed in the area pointed to by DR.

Called by: IHETSA

Entry point IHETOME

Function:

Issues WTO macro instruction if there is an abnormal KEY condition when CLOSING a file after a LOCATE statement. The file may be INDEXED (with RKP ≠ 0) or REGIONAL.

Linkage: as for IHETOMA

Called by: IHEOCL, IHEOCT

IHETPB

Entry point: IHETPBA

Function:

PRIORITY built-in function: returns the priority of a named task relative to the priority of the current task.

Linkage:

RA: A(Parameter list)
Parameter list:
A(Task variable)
A(Fullword target field)

Called by: Compiled code

IHETPR

Calls: Supervisor (CHAP,POST,WAIT)

Entry point: IHETPRA

Function:

PRIORITY pseudo-variable (PRIORITY(v) = expression): sets the priority of the specified task to the given value relative to the priority of the current task.

Linkage:

RA: A(Parameter list)
Parameter list:
A(Task variable), or zero (if current task)
A(Relative priority)

Called by: Compiled code

IHETSA

Calls:

Supervisor (ATTACH, CHAP, DEQ, DETACH, EXTRACT, FREEMAIN, GETMAIN, IDENTIFY, LINK, POST, SPIE, WAIT, WTO), IHEBEG, IHEERR, IHEMAI, IHEOCL, IHETEX

Function:

Object program management in a multitasking environment.

Entry point IHETSAA

Function:

1. Obtains storage for the PRV VDA, task variable, and event variable for the major task, and for the STOP ECB, message ECB and pointer to the chain of ECBs of the message tasks.

2. Attaches the PL/I major task and then enters a wait state until either the event variable for the major task or the STOP ECB is completed, or an abnormal task termination message is to be printed.

The execution of IHETSAA is termed the control task. Return is made to the calling program when files have been closed and storage released. (See IHETSAE.)

Linkage:

L(PRV) from linkage editor
L(LWS) from assembly of IHELIB

Called by:

Program that calls the PL/I program.

Entry point IHETSAC

Function:

To place the return code in the pseudo-register IHEQRTC.

Linkage:

RA: A(Parameter list)
Parameter List:
A(Return code) (The return code is fixed binary with default precision.)

Called by: Compiled code

Entry point IHETSAD (Get DSA)

Function:

To provide a DSA for a procedure or begin block and to set DR to point to it.

Linkage:

R0: Length of DSA
DR: A(Current save area)

Called by: Prologues

Entry point IHETSAE (END)

Function:

Frees the DSA current at entry and its associated VDAs, and abnormally terminates any tasks attached in the block. A request to free the first DSA in a subtask results in the closing of all files opened, the dequeuing of resources enqueued, and the release of all dynamic storage allocated in that task. A request to free the DSA of the main procedure also raises the FINISH condition, but does not cause controlled storage allocated in the major task to be freed.

Linkage: None

Called by: Epilogues

Entry point IHETSAF (Free VDA/LWS)

Function:

Frees the VDA or LWS at the end of the DSA chain.

Linkage:

IHEQSLA: A(VDA or LWS to be freed)
Only the most recently allocated VDA or LWS can be freed.

Called by: Compiled code, library modules

Entry point IHETSAG (GO TO)

Function:

The DSA indicated by the invocation count, or pointed to by DR, is made current. All chain elements up to this DSA, with the exception of its VDAs and itself, are freed. Any active tasks attached to the DSAs freed are abnormally terminated.

Linkage:

RA: A(Eight-byte word-aligned parameter list)

Parameter list:

```
Word 1=either  Invocation count (sign
               bit of word 2=0)
       or  PR  offset (sign bit of
               word 2=1)

Word 2=A(Location to which control is
       to be returned)
```

Called by: Compiled code

Entry point IHETSAL (Get LWS)

Function:

To provide a new LWS, and to update the LWS pseudo-registers.

Linkage: None

Called by: Library modules

Entry point IHETSAM

Function:

Initializes the PRV and primary LWS for the major task. Issues a SPIE macro instruction and branches to the main procedure.

Linkage:

RA: A(Parameter list)
Parameter list contains control information from the control task.

Attached by:

| IHETSAA, IHETSAP

Entry point IHETSAN

Function:

To change the environment of a program to that which existed at the time of

1. the execution of an ON statement associated with the on-unit to be entered, or

2. the passing of the entry parameter associated with the called procedure.

Then to branch to the on-unit or the procedure.

Linkage:

RA: A(Parameter list)
Parameter list:
A(Entry parameter). The entry parameter is an 8-byte field containing:

```
1st word: On-unit or entry address

2nd word: Invocation count of the DSA
          associated with either the
          passing procedure or the
          procedure in which the ON
          statement was executed
```

Called by: Compiled code, IHEERR

Entry point IHETSAP

Function:

As IHETSAA, but also passes a PARM parameter from the the EXEC card.

Linkage:

L(PRV) from linkage editor
L(LWS) from assembly of IHELIB

Called by: Initial entry

Entry point IHETSAR (RETURN)

Function:

Frees all chain elements up to and including the last procedure DSA in the chain. Terminates the main procedure and subtasks as in IHETSAE.

Linkage: None

Called by: Epilogues

Entry point IHETSAS

Function:

1. Allocates storage for a subtask's PRV VDA, and copies into it the PRV of the attaching task, any ON fields in the attaching DSA, and the argument list created by compiled code.

2. Issues a SPIE macro instruction and branches to the called procedure.

Linkage:

RA: A(Parameter list)
Parameter list:
A(Task variable) (Byte 0 = X '80' if no PRIORITY option; bytes 1 - 3 = 0 if no TASK option)
A(Event variable)(Zero if no EVENT option)
Relative priority
A(called procedure)
A(PRV of attaching task)
A(DSA of attaching task)
Argument list for called procedure (omitted if no argument list)

Attached by: IHETSAT

Entry point IHETSAT

Function:

To implement a CALL statement with a task option:

1. Initializes the subtask's task and event variables.

2. Attaches the subtask initialization routine (IHETSAS).

Linkage:

RA: A(Parameter list)
Parameter list:
A(Task variable) (Byte 0 = X'80' if no PRIORITY option; bytes 1 - 3 = 0 if no TASK option)
A(Event variable) (Zero if no EVENT option)
Relative priority
A(Called procedure)
Reserved
Reserved (X'80' if no argument list)
Variable length argument list for called procedure (Omitted if no argument list: X'80' in first byte of last word indicates end of list.)

Called by: Compiled code

Entry point IHETSAV (Get VDA)

Function:

To get a VDA for compiled code; sets RA=A(VDA).

Linkage:

R0: Length of VDA (excluding control words)
DR: A(Current save area)

Called by: Compiled code

Entry point IHETSAW (Get Library VDA)

Function:

To provide a VDA for library modules and to set RA = A(VDA)

Linkage:

R0: Length of VDA (including control words)

Called by: Library modules

Entry point IHETSAX

Function:

End-of-task exit routine (ETXR): detaches the TCB of a PL/I terminated task. If the task is abnormally terminated by the operating system, the control task is posted (by the POST macro) in order to print a message on SYSPRINT.

Linkage: None

Called by: Supervisor

Entry point IHETSAY

Function:

Completes the implementation of STOP: closes all opened files, releases dynamic storage, and posts the STOP ECB to cause control to return to the control task.

Linkage:

RA: Return code

Called by: IHEDUM, IHETSS

Entry point IHETSAZ

Function:

Abnormal end of task: closes all files opened in task, releases dynamic storage, and terminates the task and all subtasks attached by it.

Linkage:

RA: Return code

Called by: IHEDUM, IHEERR, IHETSE

IHETSE

Calls: IHEERR, IHETSA

Entry point: IHETSEA

Function:

To abnormally terminate the current task, and to raise the FINISH condition if the current task is the major task.

Linkage: None

Called by: Compiled code

IHETSS

Calls: IHEERR, IHETSA

Entry point: IHETSSA

Function:

To raise the FINISH condition and abnormally terminate the PL/I program in a multitasking environment.

Linkage: None

Called by: Compiled code

IHETSW

Calls:

| Supervisor (CHAP, FREEMAIN, POST, WAIT), IHEJXI, IHETSA, the I/O transmission module whose address is in the FCB.

Entry point IHETSWA

Function:

To determine whether a specified number of events has occurred. If not, to wait until the required number is complete, and, in the case of I/O events, to branch to the I/O transmission module (which raises I/O conditions if necessary). This module is used in a multitasking environment.

Linkage:

RA: A(parameter list)
Parameter list:

Word 1:

1. If all events are to be waited on:

   Byte 0 = X'FF'
   Bytes 1-3 not used

2. If a specified number (N) of events is to be waited on:

   Byte 0 = X'00'
   Bytes 1-3 = A(N)

Subsequent words (one for each element or array event):

1. Array event:

   Byte 0 = dimensionality
   Bytes 1-3 = A(ADV)

2. Element event:

   Byte 0 = X'00'
   Bytes 1-3 = A(EVENT variable)

(The high-order byte of the last argument indicates the end of the parameter list.)

Called by: Compiled code

IHEUPA

Entry Point IHEUPAA

Function:

To zero the real part of a complex coded data item and to return the address of the imaginary part.

Linkage:

RA: A(Source)
RB: A(Source DED)
WRCD: A(Imaginary part)

Called by: IHEDCN

Entry Point IHEUPAB:

Function:

To return the address of the imaginary part of a complex coded data item if switch is on, and to zero the imaginary part if switch is off.

Linkage:

RA: A(Source)
RB: A(Source DED)
WSWA: Switch for update address only
WRCD: A(Imaginary part)

Called by:

IHEDBN, IHEDCN, IHEDIA, IHEDID, IHEDIE, IHEDNC, IHEDOM, IHEVCS

IHEUPB

Calls: IHEDMA

Entry Point IHEUPBA:

Function:

To zero the real part of a complex numeric field and to return the address of the imaginary part.

Linkage:

RA: A(Source)
RB: A(Source DED)
WRCD: A(Imaginary part)

Called by: IHEDCN

Entry Point IHEUPBB:

Function:

To return the address of the imaginary part of a complex numeric field if switch is on, and to zero the imaginary part if switch is off.

Linkage:

RA: A(Source)
RB: A(Source DED)
WSWA: Switch for update address only
WRCD: A(Imaginary part)

Called by:

IHEDBN, IHEDCN, IHEDIA, IHEDID, IHEDIE, IHEDOM

IHEVCA

Entry Point: IHEVCAA

Function:

To define the attributes of arithmetic data in character form by producing a DED (flags, p, q).

Linkage:

RA: A(Target DED)
WNCP: A(Start and end addresses of data to be analysed)

Called by:

IHEDIA, IHEDIM, IHEDOM, IHELDI

IHEVCS

Calls:

IHEDMA, IHEDNB, IHEDNC, IHEUPA, IHEUPB

Entry point IHEVCSA

Function:

To direct the conversion of character representation of complex data to internal string data. The character data is first converted to coded complex, with attributes derived from the real and imaginary parts of the source data (according to the arithmetic conversion package rules) and then converted to string.

Linkage:

RA: A(Parameter list)
Parameter list:
A(Start and end addresses of real data)
A(Real DED)
A(Start and end addresses of imaginary data)
A(Imaginary DED)
A(Target SDV)
A(Target DED)
A(Real FED)
A(Imaginary FED).

Called by: IHEDIM, IHEDOM, IHELDI

Entry point IHEVCSB

Function:

As for IHEVCSA but the conversion is to coded complex only.

Linkage: As for IHEVCSA

Called by: As for IHEVCSA

IHEVFA

Calls:

IHEVKF, IHEVKG, IHEVPB, IHEVPC, IHEVPD

Entry point: IHEVFAA

Function:

Radix conversion: binary to decimal
To convert long floating-point to packed decimal intermediate.

Linkage:

WINT: Long precision floating-point number

Called by: IHEVFD, IHEVFE, IHEVPG, IHEVPH

IHEVFB

Entry point: IHEVFBA

Function:

To convert a long precision floating-point number to a fixed-point binary number with specified precision and scale factor.

Linkage:

WINT: Long precision floating-point number
WRCD: A(Target)
A(Target DED)

Called by:

IHEVFD, IHEVFE, IHEVPA, IHEVPG, IHEVPH

IHEVFC

Entry point: IHEVFCA

Function:

To convert a long floating-point number to a floating-point variable with specified precision.

Linkage:

WINT: Long-precision floating-point number
WRCD: A(Target)
A(Target DED)

Called by:

IHEVFD, IHEVFE, IHEVPA, IHEVPG, IHEVPH

IHEVFD

Calls: IHEVFA, IHEVFB, IHEVFC

Entry point: IHEVFDA

Function:

To convert a fixed-point binary integer with scale factor to long precision floating-point.

Linkage:

RA: A(Source)
RB: A(Source DED)

Called by: IHEDMA

IHEVFE

Calls: IHEVFA, IHEVFB, IHEVFC

Entry point: IHEVFEA

Function:

To convert a floating-point number of specified precision to long precision floating-point.

Linkage:

RA: A(Source)
RB: A(Source DED)

Called by: IHEDMA

IHEVKB

Calls:

IHEVKF, IHEVKG, IHEVPA, IHEVPB, IHEVPC, IHEVPD

Entry point: IHEVKBA

Function:

To convert a fixed- or floating-point decimal numeric field to packed decimal intermediate.

Linkage:

RA: A(Source)
RB: A(Source DED)

Called by: IHEDMA

<u>IHEVKC</u>

Calls:

IHEVKF, IHEVKG, IHEVPA, IHEVPB, IHEVPC, IHEVPD

Entry point: IHEVKCA

Function:

To convert a sterling numeric field to packed decimal intermediate.

Linkage:

RA: A(Source)
RB: A(Source DED)

Called by: IHEDMA

<u>IHEVKF</u>

Entry point: IHEVKFA

Function:

To convert packed decimal intermediate to a decimal fixed- or floating-point numeric field with specified precision.

Linkage:

WINT: Decimal integer
WSCF: Scale factor
WRCD: A(Target)
      A(Target DED)

Called by:

IHEVFA, IHEVKB, IHEVKC, IHEVPE, IHEVPF

<u>IHEVKG</u>

Entry point: IHEVKGA

Function:

To convert packed decimal intermediate to a sterling numeric field with specified precision.

Linkage:

WINT: Decimal integer
WSCF: Scale factor
WRCD: A(Target)
      A(Target DED)

Called by:

IHEVFA, IHEVKB, IHEVKC, IHEVPE, IHEVPF

<u>IHEVPA</u>

Calls: IHEVFB, IHEVFC

Entry point: IHEVPAA

Function:

<u>Radix conversion</u>: <u>decimal to binary</u>
To convert packed decimal intermediate to long precision floating-point.

Linkage:

WINT: Decimal integer
WSCF: Scale factor

Called by: IHEVKB, IHEVKC, IHEVPE, IHEVPF

<u>IHEVPB</u>

Entry Point: IHEVPBA

Function:

To convert packed decimal intermediate to an F format item.

Linkage:

WINT: Decimal integer
WSCF: Scale factor
WFDT: A(FED)
WRCD: A(Target)

Called by:

IHEVFA, IHEVKB, IHEVKC, IHEVPE, IHEVPF

<u>IHEVPC</u>

Entry point: IHEVPCA

Function:

To convert packed decimal intermediate to an E format item.

Linkage:

WINT: Decimal integer
WSCF: Scale factor
WFDT: A(FED)
WRCD: A(Target)

Called by:

IHEVFA, IHEVKB, IHEVKC, IHEVPE, IHEVPF

<u>IHEVPD</u>

Entry point: IHEVPDA

Function:

To convert packed decimal intermediate to a decimal integer with specified precision and scale factor.

Linkage:

```
WINT: Decimal integer
WSCF: Scale factor
WRCD: A(Target)
      A(Target DED)
```

Called by:

IHEVFA, IHEVKB, IHEVKC, IHEVPE, IHEVPF

IHEVPE

Calls:

IHEVKF, IHEVKG, IHEVPA, IHEVPB, IHEVPC, IHEVPD

Entry point: IHEVPEA

Function:

To convert an F/E format item to packed decimal intermediate.

Linkage:

```
RA: A(Source)
RB: A(Source DED)
WFED: A(FED)
```

Called by: IHEDMA

IHEVPF

Calls:

IHEVKF, IHEVKG, IHEVPA, IHEVPB, IHEVPC, IHEVPD

Entry point: IHEVPFA

Function:

To convert a decimal integer with specified precision and scale factor to packed decimal intermediate.

Linkage:

```
RA: A(Source)
RB: A(Source DED)
```

Called by: IHEDMA

IHEVPG

Calls: IHEVFA, IHEVFB, IHEVFC

Entry point: IHEVPGA

Function:

To convert a binary fixed- or floating-point constant to long precision floating-point.

Linkage:

```
WCNP: A(Beginning of constant)
      A(End of constant)
```

Called by: IHEDMA

IHEVPH

Calls: IHEVFA, IHEVFB, IHEVFC

Entry point: IHEVPHA

Function:

To convert a bit string constant with up to 31 significant bits to long precision floating-point.

Linkage:

```
WCN1: A(Beginning of constant)
      A(End of constant)
```

Called by: IHEDMA

IHEVQA

Entry point: IHEVQAA

Function:

To convert a floating point number of specified precision to a fixed-point binary number with specified precision and scale factor.

Linkage:

```
RA: A(Source)
RB: A(Source DED)
RC: A(Target)
RD: A(Target DED)
```

Called by: Compiled code, IHEVQB

IHEVQB

Calls: IHEVQA

Entry point: IHEVQBA

Function:

To convert a decimal constant to a coded arithmetic data type.

Linkage:

RA: A(First character of constant)
RB: A(Last character of constant)
RC: A(Target)
RD: A(Target DED)
WFED: A(FED) if constant is part of F or E format input
WSWB: Switches specifying type of source string

Called by: IHEDCN, IHEDIA

IHEVQC

Calls: IHEVSC, IHEVSE

Entry point: IHEVQCA

Function:

To convert some coded arithmetic data types to F or E format or character string.

Linkage:

RA: A(Source)
RB: A(Source DED)
RC: A(Target SDV)
RD: A (Target DED)
WFDT: A(FED)
WSWB: Switches specifying type of target string

Called by: IHEDNC, IHEDOA

IHEVSA

Entry point: IHEVSAA

Function:

To assign a fixed-length or VARYING bit string to a fixed-length or VARYING bit string.

Linkage:

RA: A(Source SDV)
RB: A(Source DED)
RC: A(Target SDV)
RD: A(Target DED)

Called by: Compiled code, IHEDIA, IHEDNB

IHEVSB

Entry point: IHEVSBA

Function:

To convert a fixed-length or VARYING bit string to a fixed-length or VARYING character string.

Linkage:

RA: A(Source SDV)
RB: A(Source DED)
RC: A(Target SDV)
RD: A(Target DED)

Called by:

Compiled code, IHEDOB, IHEDOD, IHEDOE, IHELDO

IHEVSC

Entry point: IHEVSCA

Function:

To assign a fixed-length or VARYING character string to a fixed-length or VARYING character string.

Linkage:

RA: A(Source SDV)
RB: A(Source DED)
RC: A(Target SDV)
RD: A(Target DED)

Called by:

Compiled code, IHEDIA, IHEDIB, IHEDID, IHEDIE, IHEDOB, IHEDOD, IHEDNC, IHELDI, IHEVQC

IHEVSD

Entry point IHEVSDA

Function:

To convert a fixed-length or VARYING character string to a fixed-length or VARYING bit string. The ONSOURCE address is stored.

Linkage:

RA: A(Source SDV)
RB: A(Source DED)
RC: A(Target SDV)
RD: A(Target DED)
WODF: A(Source SDV)

Called by:

Compiled code, IHEDIB, IHEDID, IHEDIE, IHELDI

Entry point IHEVSDB

Function:

As for IHEVSDA, but the ONSOURCE address is not stored.

Linkage:

As for IHEVSDA, but without WODF

Called by: As for IHEVSDA

IHEVSE

Entry point IHEVSEA

Function:

To assign a fixed-length or VARYING character string to a pictured character string. The ONSOURCE address is stored.

Linkage:

```
RA: A(Source SDV)
RB: A(Source DED)
RC: A(Target SDV)
RD: A(Target DED)
WODF: A(Source SDV)
```

Called by:

Compiled code, IHEDIB, IHEDID, IHEDIE, IHEDNC, IHEDOB, IHEVQC

Entry point IHEVSEB

Function:

As for IHEVSEA, but the ONSOURCE address is not stored.

Linkage:

As for IHEVSEA, but without WODF

Called by: As for IHEVSEA

IHEVSF

Entry Point: IHEVSFA

Function:

To convert a fixed-length or VARYING bit string to a pictured character string.

Linkage:

```
RA: A(Source SDV)
RB: A(Source DED)
RC: A(Target SDV)
RD: A(Target DED)
```

Called by: Compiled code, IHEDOB

IHEVTB

Base address of table: IHEVTBA

Function:

This module is a table of long precision floating-point numbers representing powers of 10 from 1 to 70. It is used by the two radix conversion routines IHEVPA and IHEVFA.

Linkage:

Not called. Referenced as external data by IHEVPA and IHEVFA

IHEXIB

Entry point: IHEXIB0

Function:

x**n, where x is real fixed-point binary and n is a positive integer.

Linkage:

```
 RA: A(x)
*RB: A(DED for x)
 RC: A(n)
 RD: A(Target)
*RE: A(Target DED)
```

Called by: Compiled code

IHEXID

Entry point: IHEXID0

Function:

x**n, where x is real fixed-point decimal, and n is a positive integer.

Linkage:

```
RA: A(x)
RB: A(DED for x)
RC: A(n)
RD: A(Target)
RE: A(Target DED)
```

Called by: Compiled code

IHEXIL

Entry point: IHEXIL0

Function:

x**n, where x is real long floating-point, and n is an integer.

Linkage:

```
RA: A(x)
RB: A(n)
RC: A(Target)
```

Called by: Compiled code

IHEXIS

Entry point: IHEXIS0

Function:

x**n, where x is real short floating-point, and n is an integer.

Linkage:

```
  RA: A(x)
  RB: A(n)
  RC: A(Target)
```

Called by: Compiled code

IHEXIU

Calls: IHEMZU

Entry point: IHEXIU0

Function:

z**n, where z is complex fixed binary and n is a positive integer.

Linkage:

```
  RA: A(z)
 *RB: A(DED for z)
  RC: A(n)
  RD: A(Target)
 *RE: A(Target)
```

Called by: Compiled code

IHEXIV

Calls: IHEMZV

Entry point: IHEXIV0

Function:

z**n, where z is complex fixed-point decimal and n is a positive integer.

Linkage:

```
  RA: A(z)
  RB: A(DED for z)
  RC: A(n)
  RD: A(Target)
 *RE: A(Target DED)
```

Called by: Compiled code

IHEXIW

Calls: IHEMZW

Entry point: IHEXIW0

Function:

z**n, where z is complex short floating-point, and n is an integer.

Linkage:

```
  RA: A(z)
  RB: A(n)
  RC: A(Target)
```

Called by: Compiled code

IHEXIZ

Calls: IHEMZZ

Entry point: IHEXIZ0

Function:

z**n, where z is complex long floating-point, and n is an integer.

Linkage:

```
  RA: A(z)
  RB: A(n)
  RC: A(Target)
```

Called by: Compiled code

IHEXXL

Calls: IHEEXL, IHELNL

Entry point: IHEXXL0

Function:

x**y, where x and y are real long floating-point.

Linkage:

```
  RA: A(y)
  RB: A(x)
  RC: A(Target)
```

Called by: Compiled code

IHEXXS

Calls: IHEEXS, IHELNS

Entry point: IHEXXS0

Function:

x**y, where x and y are real short floating-point.

Linkage:

RA: A(y)
RB: A(x)
RC: A(Target)

Called by: Compiled code

IHEXXW

Calls: IHEEXW, IHELNS, IHELNW

Entry point: IHEXXW0

Function:

z_1**z_2, where z_1 and z_2 are complex short floating-point.

Linkage:

RA: A(z_2)
RB: A(z_1)
RC: A(Target)

Called by: Compiled code

IHEXXZ

Calls: IHEEXZ, IHELNL, IHELNZ

Entry point: IHEXXZ0

Function:

z_1**z_2, where z_1 and z_2 are complex long floating-point.

Linkage:

RA: A(z_2)
RB: A(z_1)
RC: A(Target)

Called by: Compiled code

IHEYGF

Calls: IHEDMA

Entry point IHEYGFV

Function:

POLY (A,X) for both A and X vectors of real fixed-point binary or decimal numbers. Result is real short or long floating-point.

Linkage:

RA: A(Parameter list)
Parameter list:
A(ADV of argument 1)
A(DED of argument 1)
A(ADV of argument 2)
A(DED of argument 2)
A(Target)
A(DED of target)

Called by: Compiled code

Entry point IHEYGFS

Function:

As for IHEYGFV but X is scalar.

Linkage:

RA: A(Parameter list)
Parameter list:
A(ADV of argument 1)
A(DED of argument 1)
A(Argument 2)
A(DED of argument 2)
A(Target)
A(DED of target)

Called by: Compiled code

IHEYGL

Entry point IHEYGLV

Function:

POLY (A,X) for both A and X vectors of real long floating-point numbers. Result is real long floating-point.

Linkage:

RA: A(Parameter list)
Parameter list:
A(ADV of argument 1)
A(ADV of argument 2)
A(Target)

Called by: Compiled code

Entry point IHEYGLS

Function:

As for IHEYGLV but X is scalar.

Linkage:

RA: A(Parameter list)
Parameter list:
A(ADV of argument 1)
A(Argument 2)
A(Target)

Called by: Compiled code

IHEYGS

Entry point IHEYGSV

Function:

POLY (A,X) for both A and X vectors of real short floating-point. Result is real short floating-point.

Linkage:

```
RA: A(Parameter list)
Parameter list:
  A(ADV of argument 1)
  A(ADV of argument 2)
  A(Target)
```

Called by: Compiled code

Entry point IHEYGSS

Function:

As for IHEYGSV but X is scalar.

Linkage:

```
RA: A(Parameter list)
Parameter list:
  A(ADV of argument 1)
  A(Argument 2)
  A(Target)
```

Called by: Compiled code

IHEYGW

Entry point IHEYGWV

Function:

POLY (A,X) for both A and X vectors of complex short floating-point. Result is complex short floating-point.

Linkage:

```
RA: A(Parameter list)
Parameter list:
  A(ADV of argument 1)
  A(ADV of argument 2)
  A(Target)
```

Called by: Compiled code

Entry point IHEYGWS

Function:

As for IHEYGWV, but X is scalar.

Linkage:

```
RA: A(Parameter list)
Parameter list:
  A(ADV of argument 2)
  A(Argument 1)
  A(Target)
```

Called by: Compiled code

IHEYGX

Calls: IHEDMA

Entry point IHEYGXV

Function:

POLY (A,X) for both A and X vectors of complex fixed-point binary or decimal numbers. Result is complex short or long floating-point.

Linkage:

```
RA: A(Parameter list)
Parameter list:
  A(ADV of argument 1)
  A(DED of argument 1)
  A(ADV of argument 2)
  A(DED of argument 2)
  A(Target)
  A(DED of target)
```

Called by: Compiled code

Entry point IHEYGXS

Function:

As for IHEYGXV, but X is scalar.

Linkage:

```
RA: A(Parameter list)
Parameter list:
  A(ADV of argument 1)
  A(DED of argument 1)
  A(Argument 2)
  A(DED of argument 2)
  A(Target)
  A(DED of target)
```

Called by: Compiled code

IHEYGZ

Entry point IHEYGZS

Function:

As for IHEYGZV, but X is scalar.

Linkage:

RA: A(Parameter list)
Parameter list:
A(ADV of argument 1)
A(Argument 2)
A(Target)

Called by: Compiled code

Entry point IHEYGZV

Function:

POLY (A,X) for both A and X vectors of complex long floating-point numbers. Result is complex long floating-point.

Linkage:

RA: A(Parameter list)
Parameter list:
A(ADV of argument 1)
A(ADV of argument 2)
A(Target)

Called by: Compiled code

IHEZZC

Calls: IHEZZF

Entry point: IHEZZCA

Function:

To provide a SNAP dump with save-area trace and information about the PL/I files that are open.

Linkage:

RA: A(Parameter list)
See source listing for parameter list.

Called by: IHEDUM

IHEZZF

Entry point: IHEZZFA

Function:

To provide the save-area trace that forms part of the output produced by IHEZZC.

Linkage:

RA: A(Parameter list)
See source listing for parameter list.

Called by: IHEZZC

APPENDIX A: SYSTEM MACRO INSTRUCTIONS

The following table lists the system macro instructions used by the PL/I library and associates their use with individual library modules.

System Macro	Library Module
ABEND	IHEDUM, IHEERR
ATTACH	IHETSA
CHAP	IHECTT, IHEDSP, IHEIGT, IHEITB, IHEITC, IHEITH, IHEITJ, IHEOCT, IHETEA, IHETEV, IHETPR, IHETSA, IHETSW
CHECK	IHEITF, IHEITJ, IHEOPZ, IHEITB, IHEITC
CLOSE	IHECTT, IHECLS, IHECLT, IHEOPZ
DCB	IHEOPO, IHEOPZ
DCBD	IHECLT, IHECTT, IHEITB, IHEITC, IHEITD, IHEITE, IHEITF, IHEITG, IHEITH, IHEITJ, IHEOCL, IHEOCT, IHEOPO, IHEOPP, IHEOPQ, IHEOPZ
DELETE	IHECLT, IHECTT, IHEESM, IHETEX
DEQ	IHECTT, IHEDDT, IHEESM, IHEIBT, IHEITH, IHEITJ, IHEOCT, IHEPTT, IHETSA, IHETEX
DETACH	IHETSA
DEVTYPE	IHEOPO
ENQ	IHEDDT, IHEESM, IHEIBT, IHEITH, IHEITJ, IHEOCT, IHEPTT, IHETEX
ESETL	IHEITD
EXTRACT	IHETSA, IHETEX, IHETOM, IHEPRT, IHEPTT
FREEMAIN	IHEBEG, IHECLT, IHECTT, IHEDSP, IHEIOG, IHEITB, IHEITC, IHELSP, IHEMSW, IHEOCL, IHEOPZ, IHEOSW, IHESA, IHETCV, IHETSA, IHESRT, IHETSW
FREEPOOL	IHECLT, IHECTT, IHEOPQ, IHEOPZ
GET	IHEITD, IHEITG
GETBUF	IHEOPZ
GETMAIN	IHEBEG, IHEDSP, IHEERR, IHEIGT, IHEIOG, IHEITB, IHEITC, IHEITD, IHEITE, IHEITF, IHEITH, IHEITJ, IHELSP, IHEOPO, IHEOPP, IHEOPQ, IHEOPZ, IHESA, IHETCV, IHESRT, IHETSA
GETPOOL	IHEOPP
IDENTIFY	IHETSA
LINK	IHEBEG, IHEDUM, IHEERR, IHEOCL, IHEOCT, IHEOPN, IHESRT, IHETSA
LOAD	IHEESM, IHEOPQ, IHETEX
OPEN	IHEOPP, IHEOPZ
POST	IHEDSP, IHEIGT, IHEINT, IHEITB, IHEITH, IHEITJ, IHEOCT, IHETEA, IHETEV, IHETPR, IHETSA, IHETSW

PUT	IHEITD, IHEITG, IHETEX
PUTX	IHEITD, IHEITG
READ	IHEITB, IHEITE, IHEITF, IHEITH, IHEITJ
RETURN	IHECLT, IHECTT
SETL	IHEITD
SNAP	IHEDUM
SPIE	IHEERR, IHESA, IHESRT, IHETSA
STIMER	IHEOSI
TIME	IHEOSD, IHEOST
WAIT	IHEDSP, IHEIGT, IHEINT, IHEITB, IHEITE, IHEITH, IHEMSW, IHEOCT, IHEOSW, IHETEA, IHETEV, IHETPR, IHETSA, IHETSW
WRITE	IHEITB, IHEITC, IHEITE, IHEITF, IHEITH, IHEITJ, IHEOPZ
WTO	IHEDSP, IHEOCL, IHEOCT, IHEPRT, IHETOM, IHETEX, IHEPTT
WTOR	IHEDSP
XCTL	IHEOPN, IHEOPO, IHEOPP

APPENDIX B: SYSTEM GENERATION

System Generation Process

IBM System/360 Operating System consists of libraries of program modules that can be united in a variety of combinations, according to options specified by the user. The user selects the programming options that meet his data processing requirements and conform to his machine facilities. The selected options are translated into program module requirements by the system generation process, the modules being compiled into libraries that form the new operating system.

The operating system is generated in two stages. First, a series of user-supplied macro instructions, which describe the machine facilities and programming options required, is written. From these, if no errors are found, a job stream is generated. In the next stage, the job stream is processed by the assembler, the linkage editor, and utility programs, to generate the libraries of modules which form the new operating system. The whole process is carried out using an existing operating system. The system generation process is described in IBM System/360 Operating System: System Generation.

PL/I Library System Generation

All PL/I Library modules are in load form. Before system generation they exist on two libraries:

1. SYS1.PL1LIB. This PDS contains modules which are always required by a system using PL/I.
2. SYS1.LM512. This contains both modules which are optionally required and modules which will be copied into SYS1.LINKLIB.

Three PL/I Library system macros are used, whose purpose is to produce COPY control cards for inclusion in the job stream.

The first macro, SGIHE5LA, produces COPY control cards to copy modules from SYS1.LM512 into SYS1.LINKLIB.

The second macro, SGIHE5PB, produces COPY control cards to copy the non-optional modules on SYS1.PL1LIB into the new SYS1.PL1LIB.

The third macro, SGIHE5PC, tests for the COMPLEX arithmetic option. If it is present, COPY control cards are produced for modules dealing with complex arithmetic (about 30% of the total number). The macro then tests to see if the TIME and STIMER options have been requested and are available. If so, COPY control cards are produced for IHEOST and IHEOSI. If either or both of these options are not required, either or both of the dummy modules IHEMST and IHEMSI are renamed IHEOST and IHEOSI respectively and the appropriate COPY control cards are produced. Similarly, if the MULTIPLE WAIT option is not requested, the SINGLE WAIT module IHEMSW is renamed IHEOSW.

PL/I object programs require pseudo-registers (symbolic name format IHEQxxx), some of which are defined by the compiled program, others by the library modules. During execution of a program register PR always points to the base of the PRV (see 'Pseudo-Register Vector', Chapter 2).

IHEQADC

Pointer to a list of address constants for use by the I/O routines: for non-multitasking the list is in IHESA, for multitasking in IHETSA.

IHEQATV

Contains the address of the task variable for the current task.

IHEQCFL

The current-file pseudo-register, 8-bytes, word aligned. Used by STREAM I/O modules for implicit communication of the file currently being operated upon; see 'Current File' in Chapter 3.

IHEQCTS

Contains the address of the save area for the control task in a multiprogramming environment.

IHEQERR

Serves as a parameter list when calling IHEERRB. The code associated with the ON condition to be raised is placed into IHEQERR. See 'ON Conditions' in Chapter 6.

IHEQEVT

The anchor cell for the incomplete I/O event variables in a given task. When IHEQEVT contains zero, no I/O event variable in the task is incomplete.

IHEQFOP

The anchor cell of the chain linking the FCBs for the files opened in a given task. When IHEQFOP is zero, none of the files opened in this task are still open. See 'File Control Block' in Chapter 3.

IHEQFVD

Pointer to the Free VDA module: IHESAFD for non-multitasking, IHETSAF for multitasking.

IHEQINV

Contains the invocation count, and is updated by a library module each time a DSA is obtained.

IHEQLCA

Pointer to the current generation of the library communication area; see 'Library Workspace' in Chapters 2 and 4.

IHEQLSA

Pointer to the first save area in LWS, which serves two purposes: (1) the save area provided by the error-handling routines for an on-unit, and (2) an area where initial task information is saved (PICA, program mask, etc.). See Chapter 4.

IHEQLW0, IHEQLW1, IHEQLW2, IHEQLW3, IHEQLW4

Pointers to the various levels of library workspace ; see 'Library Workspace' in Chapters 2 and 4.

IHEQLWE

Pointer to the save area and workspace used by the error-handling routines when calling other library routines (not an on-unit).

IHEQLWF

Pointer to the reserved area attached to the current LWS. Used for optimization in storage management. See 'Object-time Optimization' in Chapter 4.

IHEQRTC

Contains the return code used in the normal termination of a PL/I program. (See Chapter 4.)

IHEQSAR

Contains an environment count used by the display modification module (IHESAR) when on-units and entry parameter procedures are used in prologues and epilogues.

IHEQSFC

Pointer to free-core within first block of storage obtained by the task initialization library module (IHESA); see Chapter 4.

IHEQSLA

Pointer to the storage area most recently allocated by the storage management routines. The area may be a DSA or a VDA. See Chapter 4.

IHEQSPR

The file register for SYSPRINT, the name being standardized to allow usage of the same FCB for both the source program and the library modules. See 'Standard Files', and 'File Addressing Technique' in Chapter 3.

IHEQTIC

Contains the task invocation count, which is used in multitasking in the freeing of controlled storage.

IHEQVDA

Pointer to the Get VDA module: in non-multitasking set(in IHESAP) to IHESADF; in multitasking, set (in IHETSAM) to IHETSAW.

IHEQXLV

The anchor cell for the exclusive blocks created in a given task. When IHEQXLV contains zero, the task has no exclusive blocks.

IHELIB

Operands: None

Result:

Definitions of LWS pseudo-registers.
Lengths of save areas in LWS.
Format of the library communication area.
Definitions of save area offsets.
Definitions of standard register assignments.

IHEEVT

Operands: None

Result:

Definitions of the event variable and its flags.

IHEPRV

Operands:

A three-character code denoting the last three letters of a pseudo-register name (default: LCA)
A code denoting a general register (default: WR)
A keyword parameter OP=XX, where XX is an RX instruction (default: L)

Result:

The RX operation is performed on the pseudo-register. This macro is generally used to store the pseudo-register address in a general register.

IHESDR

Operands:

A three-character code denoting a workspace level (default: LWO)
A code denoting a general register other than register DR (default: WR)

Result:

The address of the required workspace level is put into register DR.

IHEXLV

Operands: None

Result:

Definition of exclusive block and its flags.

IHEZAP

Operands: None

Result:

Definitions of I/O pseudo-registers.
Definitions of the file control block and its flag bytes.
Definition of the declare control block.
Definitions of various I/O address constants, parameters, operations and options.
Definitions of the I/O control block and its flag bytes.
Definitions of the event variable and its flags.

IHEZZZ

Operands: DUMP/none

Result:

If the operand is omitted, or is not DUMP, a full DSECT is generated. If the operand is DUMP, only the parameter list for IHEZZC is defined as a DSECT.

Used only by IHEDUM, IHEZZC, IHEZZF.

APPENDIX E: PL/I LIBRARY INTERNAL ERROR CODES AND MESSAGES

Among the errors that occur during program execution are errors that are covered by PL/I-defined conditions. If one of these occurs, an appropriate error code is passed to IHEERR in pseudo-register IHEQERR. This code is a 4-digit hexadecimal number. The two high-order digits denote the PL/I condition (Figure 49); the others denote the errors associated with that condition.

Code	Condition
10	STRINGRANGE
18	OVERFLOW
20	SIZE
28	FIXEDOVERFLOW
30	SUBSCRIPTRANGE
38	CHECK(label)
40	CONVERSION
48	CHECK(variable)
50	CONDITION(identifier)
58	FINISH
60	ERROR
68	ZERODIVIDE
70	UNDERFLOW
78	AREA
88	NAME
90	RECORD
98	TRANSMIT
A0	I/O SIZE
A8	KEY
B0	ENDPAGE
B8	ENDFILE
C0	I/O CONVERSION
C8	UNDEFINEDFILE

Figure 49. Internal Codes for ON Condition Entries

If system action is required, an error message will be printed. The messages relating to the errors for the PL/I conditions are given here.

Error code	Message
1000	STRINGRANGE
1800	OVERFLOW
2000	SIZE
2800	FIXEDOVERFLOW
3000	SUBSCRIPTRANGE
4000	CONVERSION
4001	CONVERSION ERROR IN F-FORMAT INPUT
4002	CONVERSION ERROR IN E-FORMAT INPUT
4003	CONVERSION ERROR IN B-FORMAT INPUT
4004	ERROR IN CONVERSION FROM CHARACTER STRING TO ARITHMETIC
4005	ERROR IN CONVERSION FROM CHARACTER STRING TO BIT STRING
4006	ERROR IN CONVERSION FROM CHARACTER STRING TO PICTURED CHARACTER STRING
4007	CONVERSION ERROR IN P-FORMAT INPUT (DECIMAL)
4008	CONVERSION ERROR IN P-FORMAT INPUT (CHARACTER)
4009	CONVERSION ERROR IN P-FORMAT INPUT (STERLING)
5000	CONDITION
5800	FINISH
6000	ERROR
6800	ZERODIVIDE
7000	UNDERFLOW
7800	AREA SIGNALED
7801	AREA CONDITION RAISED IN ASSIGNMENT STATEMENT
7802	AREA CONDITION RAISED IN ALLOCATE STATEMENT
8800	UNRECOGNIZABLE DATA NAME
9000	RECORD CONDITION SIGNALED
9001	RECORD VARIABLE SMALLER THAN RECORD SIZE
9002	RECORD VARIABLE LARGER THAN RECORD SIZE
9003	ATTEMPT TO WRITE ZERO LENGTH RECORD
9004	ZERO LENGTH RECORD READ
9800	TRANSMIT CONDITION SIGNALED
9801	PERMANENT OUTPUT ERROR

9802	PERMANENT INPUT ERROR
A800	KEY CONDITION SIGNALED
A801	KEYED RECORD NOT FOUND
A802	ATTEMPT TO ADD DUPLICATE KEY
A803	KEY SEQUENCE ERROR
A804	KEY CONVERSION ERROR
A805	KEY SPECIFICATION ERROR
A806	KEYED RELATIVE RECORD/TRACK OUTSIDE DATA SET LIMIT
A807	NO SPACE AVAILABLE TO ADD KEYED RECORD
B800	END OF FILE ENCOUNTERED
C800	UNDEFINEDFILE CONDITION SIGNALED
C801	FILE ATTRIBUTE CONFLICT AT OPEN
C802	FILE TYPE NOT SUPPORTED
C803	BLOCKSIZE NOT SPECIFIED
C804	CANNOT BE OPENED (NO DD CARD)
C805	ERROR INITIALIZING REGIONAL DATA SET
C806	CONFLICTING ATTRIBUTE AND ENVIRONMENT PARAMETERS
C807	CONFLICTING ENVIRONMENT AND/OR DD PARAMETERS
C808	KEY LENGTH NOT SPECIFIED
C809	INCORRECT BLOCKSIZE AND/OR LOGICAL RECORD SIZE
C80A	LINESIZE GT IMPLEMENTATION DEFINED MAXIMUM LENGTH
C80B	CONFLICTING ATTRIBUTE AND DD PARAMETERS

APPENDIX F: DUMP INDEX

The dump index provided by the subroutines IHEZZA, IHEZZB, and IHEZZC contains information about:

SYSPRINT buffers

Files currently open

Current file

Save areas

On-units, interrupts and other details

This information is output to a file called PL1DUMP.

SYSPRINT Buffers

The contents of each buffer are given, in EBCDIC. If U-format records are used, the contents of the intermediate buffer used by the library are also printed.

Files Currently Open

File name

A(DCLCB)

A(FCB)

A(DCB)

File-register offset in PRV

Current File

I/O Files: File name

A(DCLCB)

A(FCB)

A(DCB)

STRING Files: A(SDV)

Save Areas

A trace-back through the save-area chain provides the following addresses:

A(All save areas, including the library save areas)

A(Current LCA)

A(PRV VDA)

A(VDA for LWS2)

Other Information

If a CALL was made: A(CALL)
A(Procedure) or
A(Entry point of library module)

If a BEGIN block was entered: A(Entry point)

If a program interrupt occurs: A(Interrupt)

If an on-unit was entered: Type of on-unit. If this on-unit is the error on-unit and was entered as a result of system action, the condition causing the system action is given.

If IHEDMA occurs in the trace-back: The names of the modules used in the conversion are given.

The statement number (if it exists) is given.

The following program illustrates the use of the dump index:

```
     TDUMP: PROC OPTIONS (MAIN);


 1             TDUMP: PROC OPTIONS(MAIN);
 2                    DCL   A CHAR(4)INIT('ABCD');
 3             DCL IHESARC ENTRY(FIXED BINARY);
 4                    ON ERROR CALL IHEDUMP;
 6                    ON CONV CALL CONVPROC;
 8             CALL IHESARC(20);
 9                    PUT LIST ('THIS IS THE FIRST LINE');
10                    PUT SKIP LIST('THIS IS THE SECOND
                      LINE');
11                    OPEN FILE(XYZ) OUTPUT;
12                    BEGIN;
13                    X=A;                          /* CONV ERROR */
14             END ;
15             CONVPROC:PROC;
16                    DCL   Y(-32768:-32768,-32768:-32768) CHAR(1);
17                    Z=Y(32767,32767); /* ADDRESSING ERROR */
18                    END TDUMP;
```

The addressing error only occurs if this program is the only one being executed.

The dump index produced for this program is:

```
                                  * * * PL/I F-COMPILER 4TH VERSION * IHEDUMP * * *


* * * SYSPRINT BUFFERS

BUFFER 01

                 HE FIRST LINE "          U       YA 3                  R IHEOPNA O O

BUFFER 02

        IHE804I ADDRESSING INTERRUPT IN STATEMENT 00017 AT OFFSET +000B4
                FROM ENTRY POINT CONVPROC


*** FILES CURRENTLY OPEN

XYZ                            DCLCB 00A488   FCB 03EB40   DCB 03EB70   PR OFFSET 01C
SYSPRINT                       DCLCB 00A4C0   FCB 03EBD0   DCB 03EC00   PR OFFSET 020


*** CHAIN BACK THROUGH SAVE AREAS

03F9B0  DSA FOR ERR ON-UNIT                CALLS IHEDUMP   FROM 00A1FA (STMT 5)

03DF10  SECONDARY LIBRARY WORKSPACE

03DF20  SAVE AREA FOR LIBRARY              CALLS 00A19C    FROM 00CA3E   LCA  AT 03E3]

03F690  SAVE AREA FOR LIBRARY              CALLS 00A522    FROM 00CA04   LCA AT 03F730

03F4C8  SAVE AREA FOR LIBRARY                     INTERRUPT AT 00AF46   LCA AT 03F730

03F8D8  DSA FOR PROC CONVPROC              CALLS 00AEF0    FROM 00A318 (STMT 17)

03F828  DSA FOR CONV ON-UNIT               CALLS 00A264    FROM 00A25E (STMT 7)

03F338  SECONDARY LIBRARY WORKSPACE

03F348  SAVE AREA FOR LIBRARY              CALLS 00A200    FROM 00CA3E   LCA AT 03F730

03F018  SAVE AREA FOR LIBRARY              CALLS 00A522    FROM 00CA04   LCA AT 03F0B8
```

```
03EDB8   SAVE AREA FOR LIBRARY                CALLS 00C728    FROM 00B9CA   LCA AT 03F0B8

03FE50   SAVE AREA FOR LIBRARY                CALLS 00B8D0    FROM 00AF06   LCA AT 03F0B8

03F290   DSA FOR BEGIN                        CALLS 00AEF0    FROM 00A186 (STMT 13)

03F1B0   DSA FOR PROC TDUMP                   ENTERS BEGIN AT 00A138

03EC60   PRV - PSEUDO REGISTERS START AT 03EC68

03FFB4   EXTERNAL SA                          CALLS 00A020

*** END OF OUTPUT
```

When V-format records are used, the first nine data characters of one of the SYSPRINT buffers may be blanked out.

If there had been a current file, this would have appeared after the section on 'Files Currently Opened'.

The following list comprises all the library modules provided for Version 4 of the PL/I (F) Compiler. It gives the length in bytes of each module. Some of the modules are not required by Version 4, but are included for compatibility with previous versions; numbers in parentheses after the names of these modules indicate the versions that do use them. The modules marked * reside in the link library (SYS1.LINKLIB); all other modules are in SYS1.PL1LIB.

	Module		Length
	IHEABU		184
	IHEABV		544
	IHEABW		128
	IHEABZ		128
	IHEADD		216
	IHEADV		96
	IHEAPD		360
	IHEATL		536
	IHEATS		408
	IHEATW		304
	IHEATZ		296
	IHEBEG		136
	IHEBSA		296
	IHEBSC		272
	IHEBSD		192
	IHEBSF		480
	IHEBSI		296
	IHEBSK		472
	IHEBSM		384
	IHEBSN		192
	IHEBSO		312
	IHEBSS		240
	IHECFA		160
	IHECFB		576
	IHECFC		88
	IHECKP		184
*	IHECLS	(1,2,3)	992
*	IHECLT		1298
	IHECNT		72
	IHECSC		200
	IHECSI		168
	IHECSK		320
	IHECSM		280
	IHECSS		224
	IHECTT		1718
	IHEDBN		344
	IHEDCN		495
	IHEDDI		1248
	IHEDDJ		448
	IHEDDO		648
	IHEDDP		640
	IHEDDT		760
	IHEDIA		584
	IHEDIB		280
	IHEDID		448
	IHEDIE		456
	IHEDIL		48
	IHEDIM		528
	IHEDMA		248
	IHEDNB		248
	IHEDNC		632
	IHEDOA		224
	IHEDOB		328
	IHEDOD		296
	IHEDOE		224
	IHEDOM		584
	IHEDSP		612
	IHEDUM		280
	IHEDVU		408
	IHEDVV		576
	IHEDZW		104
	IHEDZZ		104
	IHEEFL		736
	IHEEFS		384
*	IHEERD		720
*	IHEERE		1704
*	IHEERI		896
*	IHEERN	(1,2)	4096
*	IHEERO		856
*	IHEERP		1208
	IHEERR		1816
*	IHEERS	(1)	848
*	IHEERT		712
*	IHEESM		1768
*	IHEESS	(2)	1960
	IHEEXL		456
	IHEEXS		256
	IHEEXW		136
	IHEEXZ		136
	IHEHTL		272
	IHEHTS		192
	IHEIBT		576
	IHEIGT	(1,2,3,4)	1340
	IHEINT		436
	IHEIOA		360
	IHEIOB		480
	IHEIOC		288
	IHEIOD		640
	IHEIOE	(1,2,3)	176
	IHEIOF		736
	IHEIOG	(1,2,3,4)	1104
	IHEIOH	(2)	200
*	IHEIOJ	(2,3)	1992
	IHEION		248
	IHEIOP		488
	IHEIOX		328
*	IHEITB		3772
*	IHEITC		2604
*	IHEITD		2270
*	IHEITE		1760
*	IHEITF		1845
*	IHEITG		1122
*	IHEITH		2610
*	IHEITJ		2650
*	IHEITK		622
*	IHEITL		492
	IHEJXI		320
	IHEJXS		104
	IHEKCA		1560

IHEKCB	1464
IHEKCD	256
IHELDI	2072
IHELDO	1048
IHELNL	360
IHELNS	256
IHELNW	268
IHELNZ	288
IHELSP	1064
IHEM91	344
IHEMAI	8
IHEMPU	240
IHEMPV	288
IHEMSI	32
IHEMST	32
IHEMSW	136
IHEMXB	96
IHEMXD	120
IHEMXL	96
IHEMXS	96
IHEMZU	240
IHEMZV	672
IHEMZW	64
IHEMZZ	64
IHENL1	280
IHENL2	192
IHEOCL	1338
IHEOCT	2190
* IHEOPN	920
* IHEOPO	1828
* IHEOPP	1874
* IHEOPQ	1296
* IHEOPZ	992
IHEOSD	216
IHEOSE	80
IHEOSI	72
IHEOSS	56
IHEOST	88
IHEOSW	1060
IHEPDF	144
IHEPDL	88
IHEPDS	88
IHEPDW	120
IHEPDX	272
IHEPDZ	120
IHEPRT	656
IHEPSF	160
IHEPSL	72
IHEPSS	72
IHEPSW	96
IHEPSX	256
IHEPSZ	96
IHEPTT	768
IHESA	2488
IHESHL	248
IHESHS	192
IHESMF	136
IHESMG	128
IHESMH	128
IHESMX	224
IHESNL	416
IHESNS	320
IHESNW	320
IHESNZ	368
IHESQL	160
IHESQS	168
IHESQW	152
IHESQZ	144
IHESRC	344
IHESRD	128
IHESRT	1348
IHESSF	168
IHESSG	104
IHESSH	104
IHESSX	216
IHESTG	1108
IHESTR	1592
IHETAB	16
IHETCV	208
IHETEA	248
IHETER	272
IHETEV	240
* IHETEX	1464
IHETHL	280
IHETHS	200
IHETNL	344
IHETNS	272
IHETNW	184
IHETNZ	184
* IHETOM	493
IHETPB	56
IHETPR	268
IHETSA	5720
IHETSE	88
IHETSS	72
IHETSW	1520
IHEUPA	192
IHEUPB	232
IHEVCA	272
IHEVCS	480
IHEVFA	232
IHEVFB	224
IHEVFC	40
IHEVFD	88
IHEVFE	32
IHEVKB	736
IHEVKC	720
IHEVKF	1504
IHEVKG	1248
IHEVPA	352
IHEVPB	408
IHEVPC	492
IHEVPD	264
IHEVPE	616
IHEVPF	72
IHEVPG	560
IHEVPH	184
IHEVQA	208
IHEVQB	1004
IHEVQC	600
IHEVSA	320
IHEVSB	208
IHEVSC	176
IHEVSD	416
IHEVSE	352
IHEVSF	240
IHEVTB	136
IHEXIB	88
IHEXID	136
IHEXIL	152
IHEXIS	152
IHEXIU	120
IHEXIV	192
IHEXIW	256
IHEXIZ	256
IHEXXL	152

	IHEXXS	144
	IHEXXW	280
	IHEXXZ	280
	IHEYGF	432
	IHEYGL	240
	IHEYGS	240
	IHEYGW	280
	IHEYGX	688
	IHEYGZ	280
*	IHEZZA (3)	1296
*	IHEZZB (3)	1704
*	IHEZZC	2960
*	IHEZZF	1596

APPENDIX H: COMPILER-GENERATED CONTROL BLOCKS

This appendix describes all the compiler-generated control blocks used by the PL/I Library except the DCLCB and the OCB, which are described in Appendix I (Input/Output Control Blocks). All offsets are given in hexadecimal form.

0 2	3 7	8 15	16 31
BtOf		Virtual origin	
Multiplier$_1$			
. . .			
Multiplier$_n$			
Upper bound$_1$		Lower bound$_1$	
. . .		. . .	
Upper bound$_n$		Lower bound$_n$	

Figure 50. Format of the Array Dope Vector (ADV)

This control block contains information required in the derivation of elemental addresses within an array data aggregate. The ADV is used for three functions within the library:

1. Given an array, to step through the array in row-major order.

2. Given the subscript values of an array element, to determine the element address.

3. Given an element address, to determine its subscript values.

Within PL/I implementation, arrays are stored in row-major order, upward in storage. The elements of an array are normally in contiguous storage; if the array is a member of a structure, its elements may be discontiguous. Such discontiguity, however, is transparent to algorithms which employ an array dope vector.

The ADV contains (2n + 1) 32-bit words, where n is the number of dimensions of the array. The number of dimensions in the array is not described within the ADV, but is passed to the library as an additional argument.

<u>Definitions of ADV fields:</u>

BtOf (= Bit offset): For an array of bit strings with the UNALIGNED attribute, this is the bit offset from the byte address of the virtual origin.

Virtual origin: The byte address of the array element whose subscript values are all zero, i.e.,X(0,...,0);this element need not be an actual member of the array, in which case the virtual origin will address a location in storage outside the actual bounds of the array.

Multiplier: These are fullword binary integers which, in the standard ADV algorithm, effect dimensional incrementation or decrementation to locate an element. Bit multipliers are used for fixed-length bit string arrays; byte multipliers are used for everything else.

Upper Bound: Halfword binary integer, specifying the maximum value permitted for a subscript in the ith dimension. This value may be negative.

Lower Bound: Halfword binary integer, specifying the minimum value permitted for a subscript in the ith dimension. The value may be negative.

<u>ADV Algorithm:</u> Given subscript values for an n-dimensional array, the address of any element is computed as:

$$\text{Address} = \text{origin} + \sum_{i=1}^{n} S_i * M_i$$

where S_i = value of the ith subscript
M_i = value of the ith multiplier

For an array of bit strings with the UNALIGNED attribute, the origin is a bit address formed by concatenating the virual origin and the bit offset. For all other arrays, the origin is the virtual origin.

Data type	Representation	Bytes					
		1	2	3	4	5	6 and onwards
Arithmetic	Fixed-point Floating-point Packed decimal	Flags	p	q	-	-	-
	Numeric field	Flags	p	q	w	1	Picture specn
String	Unpictured	Flags	-	-	-	-	-
	Pictured	Flags	1		Picture specification		

Figure 51. Format of the Data Element Descriptor (DED)

Code	Bit							
	0	1	2	3	4	5	6	7
= 0	0 = String	*	Unaligned	Fixed	Picture	Bit	*	0
= 1		*	Aligned	Varying	No Picture	Character	*	0
= 0	1 = Arithmetic	*	Non-sterling	Short	Numeric field	Decimal	Fixed	Real
= 1		*	Sterling	Long	Coded	Binary	Float	Complex

* These bits are used by the compiler, but, when a DED is passed to a library module, they are always set to zero.

•Figure 52. Format of the DED Flag Byte

Data element descriptors (DEDs) contain information derived from explicit or implicit declarations of variables of type arithmetic and string. There are four DED formats; they are shown in Figure 51.

Definitions of DED fields:

Flags: An eight-bit encoded form of declared information (Figure 52). Those flags which are specified as zero must be set to zero.

p byte: p is the declared or default precision of the data item.

q byte: q is the declared or default scale factor of the data item, in excess-128 notation (i.e., if the implied fractional point is between the last and the next-to-last digit, q will have the value 129).

For numeric fields, q is the resultant scale factor derived from the apparent precision as specified in the picture, i.e., the number of digit positions after a V picture item as modified by an F (scale factor) item.

For fixed decimal pictures, any explicit scaling of the form F(±I) is combined with the implied scale, as described above, and reflected in the DED. The F(±I) is then no longer required and is removed from the picture.

w byte: w specifies the number of storage units allocated for a numeric field.

1 byte(s): 1 specifies the number of bytes allocated for the picture associated with a numeric field. If the data item is string, 1 occupies two bytes; if arithmetic, one byte.

Picture specification: This field contains the picture declared for the data item. If the data item is string, the picture may occupy 1 through 32,767 bytes; if arithmetic, 1 through 255 bytes. If the original picture specification contained replication factors, it will have been expanded in full.

DOPE VECTOR DESCRIPTOR (DVD)

This provides a key for scanning the standard array, string and structure dope vectors. It consists of one entry for each major structure, minor structure and base element in the original declaration. Each entry consists of one word and can have one of two formats:

1. Structure:

```
 0  1  2           7 8                 15
┌──┬──┬───────────┬─────────────────────┐
│F1│F2│     L     │          N          │
└──┴──┴───────────┴─────────────────────┘
 16                                    31
┌───────────────────────────────────────┐
│                Offset                 │
└───────────────────────────────────────┘
```

F1	= 0	Structure
F2	= 0	
L	=	Level of structure
N	=	Dimensionality, including inherited dimensions
Offset	=	Offset of containing structure from start of DVD
	=	- 1 for a major structure

2. Base element:

```
 0  1  2          7  8  9  10          15
┌──┬──┬──────────┬──┬──┬──────────────┐
│F1│F2│    L     │F5│F6│      N       │
└──┴──┴──────────┴──┴──┴──────────────┘
 16 17 18        23 24                31
┌──┬──┬──────────┬──┬──┬──────────────┐
│F3│F4│    A     │  │  │      D       │
└──┴──┴──────────┴──┴──┴──────────────┘
```

F1	= 1	Base element
F2	= 0	Not end of structure
	= 1	End of structure
L	=	Level of element
F5	= 1	Area variable
	= 0	Not area variable
F6	= 1	Event variable
	= 0	Not event variable
N	=	Dimensionality
F3	= 0	Not an aligned bit string
	= 1	Aligned bit string
F4	= 0	Not a varying string
	= 1	Varying string
A	=	Alignment in bits (0 to 63)
D	=	Length, if not a string, in bits
	=	0 if a string, in which case the length is in the dope vector

FORMAT ELEMENT DESCRIPTOR (FED)

This control block contains information derived from a format element within a format list specification for edit-directed I/O. There are five forms of the FED:

1. Format item E:

```
  1   2   3   4
┌───────┬───┬───┐
│   w   │ d │ s │
└───────┴───┴───┘
```

   w = width of data field in characters

   d = number of digits following decimal point

   s = number of significant digits to be placed in data field (ignored for input)

2. Format item F:

```
  1   2   3   4
┌───────┬───┬───┐
│   w   │ d │ p │
└───────┴───┴───┘
```

   w and d: as for E format

   p = scale factor in excess-128 notation

3. Format items A, B, X:

```
  1   2
┌───────┐
│   w   │
└───────┘
```

   w = as for E format

4. Format item P:

   There are two forms of the FED for the P format items, these being identical to the DEDs for numeric fields and pictured character strings.

5. Printing format items PAGE, SKIP, LINE, COLUMN:

   The FEDs for SKIP, LINE and COLUMN are halfword binary integers. PAGE does not have an FED.

	Symbolic name	Length (bytes)	Function
0	WBR1	4	2nd XCTL address for communication in arithmetic conversion package.
4	WBR2	4	3rd XCTL address for communication in arithmetic conversion package.
8	WRCD	8	A(Target),A(DED): Implicit parameters for final conversion in arithmetic scheme. Stored by arithmetic director.
10	WFED	4	A(Source FED): Implicit parameter for F or E format input conversion.
14	WSCF	4	Scale factor for library decimal intermediate form.
18	WSDV	8	Input/output field dope vector.
20	WINT	9	Library intermediate form storage area.
29	WSWA	1	Eight 1-bit switches: Intermodular communication.
2A	WSWB	1	Eight 1-bit switches: General purpose switches.
2B	WSWC	1	Eight 1-bit switches: Not used across calls.
2C	WOFD	8	Dope vector for ONSOURCE or ONKEY built-in functions.
34	WOCH	4	A(Error character): ONCHAR built-in function.
38	WFCS	150	Character string (in required format) used by list-directed and data-directed output.
CE	WCFD	4	Library intermediate FED: String/arithmetic conversion.
D2	WFDT	4	A(Target FED): Implicit parameter for F or E format output conversion.
D6	WODF	8	SDV for DATAFIELD in error.
DE	WCNV	8	Library GO TO control block.
E6	WFIL	4	A(DCLCB) for ONFILE.
EA	WOKY	8	SDV(Null string); requested when ONKEY built-in function used out of context.
F2	WEVT	4	A5(event variable).
F6	WREA	4	Return address for AREA on-unit.

Alternative entries:

38	WFC1	40	Workspace for interleaved array indexer.
60	WONC	40	Error code; storage area for contents of floating-point registers in error-handling subroutines.

38	WCNP	4	Implicit parameter: A(Constant descriptor).
3C	WCN1	8	A(Start of constant), A(End of constant).
44	WCN2	8	A(Start of constant), A(End of constant).

Figure 53. Library Communication Area (LCA)

The library communication area (LCA) is part of library workspace (LWS), the format of which is given in Figure 54. The use of LWS and LCA is described in 'Communication Conventions' in Chapter 2.

```
                         0       7 8                      31
IHEQLSA-------->┌────────┬──────────────────────────┐
              0 | Flags  |        Length            |
                ├────────┴──────────────────────────┤
              4 | Chain-back address (save area)    |
                ├───────────────────────────────────┤
              8 | Chain-forward address             |
                ├───────────────────────────────────┤
              C |                                   |
                | Register save area                |
                |                                   |
                ├───────────────────────────────────┤
             48 | (8 bytes unused)                  |
                |                                   |
IHEQLWO-------->├───────────────────────────────────┤
             50 |                                   |
                |                                   |
                | Workspace level 0                 |
                |                                   |
                |                                   |
IHEQLW1-------->├───────────────────────────────────┤
             E8 |                                   |
                |                                   |
                | Workspace level 1                 |
                |                                   |
                |                                   |
IHEQLW2-------->├───────────────────────────────────┤
            180 |                                   |
                |                                   |
                | Workspace level 2                 |
                |                                   |
                |                                   |
IHEQLW3-------->├───────────────────────────────────┤
            218 |                                   |
                |                                   |
                | Workspace level 3                 |
                |                                   |
                |                                   |
IHEQLW4-------->├───────────────────────────────────┤
            2B0 |                                   |
                |                                   |
                | Workspace level 4                 |
                |                                   |
                |                                   |
IHEQLWE-------->├───────────────────────────────────┤
            348 |                                   |
                |                                   |
                | Workspace level E                 |
                |                                   |
                |                                   |
IHEQLCA-------->├───────────────────────────────────┤
            3E0 |                                   |
                |                                   |
                |                                   |
                | Library communication area (LCA)  |
                |                                   |
                |                                   |
IHEQLWF-------->└───────────────────────────────────┘
```

Figure 54. Standard Format of Library Workspace (LWS)

The use of Library Workspace (LWS) is described in Chapter 2. The format of the LCA is given in Figure 53 and that of the SSA in Figure 55.

Offset		General Register		Standard Save Area	
				Usage	
Value	Symbolic Name	Number	Symbolic Name	0 7	8 31
0	OFCD	-	-	Flags	Length
4	OFDR	13	DR	Chain-back address	
8	-	-	-	Chain-forward address	
C	OFLR	14	LR,RY		
10	OFBR	15	BR,RZ		
14	OFR0	0	R0	Contents of register	
18	OFRA	1	R1,RA		
1C	OFRB	2	RB	Contents of register	
20	OFRC	3	RC	Contents of register	
24	OFRD	4	RD	Contents of register	
28	OFRE	5	RE	Contents of register	
2C	OFRF	6	RF	Contents of register	
30	OFRG	7	RG	Contents of register	
34	OFRH	8	RH	Contents of register	
38	OFRI	9	RI	Contents of register	
3C	OFRJ	10	RJ	Contents of register	
40	OFWR	11	RX,WR		
44	OFPR	12	PR	Pseudo-register pointer	

Figure 55. Format of the Standard Save Area (SSA)

Flags: One-byte code, employed by PL/I housekeeping procedures to specify the nature of the storage area in which the SSA resides. (See Figure 56.)

Length: Three-byte binary integer specifying the total length of the storage area in which the SSA resides; used by PL/I housekeeping to free dynamic storage areas. (See 'PL/I Object Program Management'.) When OPT=01.Default is used, bit 1 of these three bytes is used as a flag.

Chain-back Address: Address of the SSA originally provided for a module that now calls another module.

Chain-forward Address: Address of the SSA acquired by a called module. This field is not set for any PL/I Library module, since intermodule trace is not supported within the library.

Return address of the calling module: Contents of register LR on entry to the called module, set by the calling module to the address of the point of return. All PL/I Library modules return using register LR.

Entry Point of the called module: Contents of register BR on entry to the called module.

R0 to PR: Contents of the specified registers on entry to the called module. PL/I Library modules save all registers

LR through WR in order to meet the requirements of a GO TO statement in an on-unit. (See Chapter 4.) The register PR field is set by the subroutine in IHESA that initializes the main procedure; it remains unchanged throughout the task.

Bit	Meaning	
	= 0	= 1
0	Always = 1	
1	No statement number field in DSA	Statement number field in DSA
2	No dummy ON field for STRINGRANGE	STRINGRANGE field created as for other ON conditions
3	Procedure DSA	Begin block DSA
4	No dummy ON field for SUBSCRIPTRANGE	SUBSCRIPTRANGE field created as for other ON conditions
5	Non-recursive DSA, without display update field	Recursive DSA, with display update field
6	No ON fields	ON fields
7	No dummy ON field for SIZE	SIZE field created as for other ON conditions

Figure 56. Format of the SSA Flag Byte

```
 0                 15 16                  31
┌────────────────────────────────────────┐
│                                        │
│                                        │
│                                        │
│                  ADV                   │
│                                        │
│                                        │
│                                        │
├───────────────────┬────────────────────┤
│  Maximum length   │  Current length/0  │
└───────────────────┴────────────────────┘
```

Figure 57. Format of the Primary String Array Dope Vector (SADV)

This control block contains information required to derive, directly or indirectly (through a secondary array of SDV entries), the address of elemental strings. The SADV is identical to the basic ADV, with the addition of a fullword which describes the string length.

Fixed-length strings require only a primary dope vector. The two length fields are set to the same value, which is the declared length of the strings.

VARYING strings require, in addition to the primary dope vector, a secondary dope vector. This consists of SDV entries for each elemental string within the array. The secondary dope vector is addressed via the primary dope vector by the standard ADV algorithm; having located the relevant SDV, the actual string data is directly addressable. The maximum-length field appended to the ADV is set to the declared maximum length of each array element. The current-length field is set to zero.

The multipliers of the ADV for a fixed-length string apply to the actual string data. Those of the ADV for a variable-length string apply to the secondary dope vector of SDV entries.

```
 0  2 3   7 8     15 16                31
┌────┬─────┬────────────────────────────┐
│BtOf│     │   Byte address of string   │
├────┴─────┴────────┬───────────────────┤
│  Maximum length   │   Current length  │
└───────────────────┴───────────────────┘
```
Figure 58. Format of the String Dope Vector (SDV)

A string dope vector (SDV) is an 8-byte word-aligned block that specifies storage requirements for string data.

Definition of SDV fields:

BtOf (Bit offset): If the string is a bit string, positions 0 to 2 of the SDV specify the offset of the first bit of the string within the addressed byte. The bit offset is only applicable to bit strings which form part of a data aggregate, and then only if that aggregate has the UNALIGNED attribute.

Byte address of string: For both character and bit strings, this three-byte field specifies the address of the initial byte of the string.

Maximum length: Halfword binary integer which specifies the number of storage units allocated for the string; byte count if character string, bit count if bit string. This value does not vary for a particular generation of its associated string.

Current length: Halfword binary integer which specifies the number of storage units, within the maximum length, currently occupied by the string; only applicable to strings with the VARYING attribute.

The two length fields exist to accommodate strings with the VARYING attribute; in the instance of a fixed-length string, the two fields contain identical values. Both fields may contain a maximum value of 32,767.

STRUCTURE DOPE VECTOR

This control block contains information required to derive, directly or indirectly, the address of all elements of the structure.

The format of a structure dope vector is determined as follows. The dimensions which have been applied to the major structure or to minor structures are inherited by the contained structure base elements; undimensioned non-string base elements are assigned a dope vector consisting only of a single-word address field. The structure dope vector is then derived by concatenating the dope vectors which the base elements would have if they were not part of a structure, in the order in which the elements appear in the structure.

```
 0       7 8      15 16                 31
┌─────────┬──────────────────────────────┐
│    0    │    Chain-forward address     │
├─────────┼──────────────────────────────┤
│ Length  │                              │
├─────────┘                              │
│                                        │
│              Identifier                │
│                                        │
├─────────┬──────────────────────────────┤
│    D    │            A(DED)            │
├─────────┼──────────────────────────────┤
│  Flags  │           Field A            │
├─────────┴──────────┬───────────────────┤
│     Field B        │                   │
└────────────────────┴───────────────────┘
```

Figure 59. Format of the Symbol Table (SYMTAB)

The symbol table consists of one or more entries which define the attributes, identifier, and storage location of variables which appear in the data list for data-directed I/O. Each SYMTAB entry contains the address of the next entry or a stopper.

Definition of SYMTAB fields:

Chain-forward address: The address of the next entry in the symbol table; all symbols (identifiers) known within a given block are chained together. The last entry in the chain is signaled by a zero chain-forward address. (The symbol table of a contained block must include the symbol table of the containing block; hence the chain-forward address of the last entry for variables declared in a contained block is that of the first entry in the symbol table of the containing block.)

Length: Number of characters comprising the identifier. Maximum length is 255 characters.

Identifier: The name declared for a variable. If the variable is known by a qualified name, the identifier includes separating periods.

D (= Dimensionality): The number of dimensions declared for an array variable; D = 0 for scalar variables.

A(DED): Address of the data element descriptor associated with the variable.

Flags:

Bit		
0		(Reserved)
1	= 1	ON CHECK for the variable
2	= 1	ON CHECK for label variable
3		(Reserved)
4		(Reserved)

Bits

5 6 7	
0 0 0	Variable is STATIC
0 0 1	Non-structured AUTOMATIC or CONTROLLED
0 1 0	Structured AUTOMATIC or CONTROLLED

Field A:

If STATIC: Address of data item or its dope vector.

If AUTOMATIC (non-structured): Offset of data item or its dope vector within DSA. (See note.)

If AUTOMATIC (structured): Offset of dope vector for data item (within a structure dope vector), relative to origin of DSA. (See note.)

If CONTROLLED (non-structured): Offset to data item or its dope vector.

If CONTROLLED(structured): As for AUTOMATIC (structured), but offset is relative to origin of structure dope vector.

Field B:

If STATIC: Not used.

If AUTOMATIC: Offset of display within PRV.

If CONTROLLED: Offset of the anchor word (pseudo-register) of the controlled variable.

Note: See Chapter 4 for description of storage class implementation and for definition of DSA.

APPENDIX I: INPUT/OUTPUT CONTROL BLOCKS

This appendix gives the formats of the control blocks used by the PL/I Library I/O interface modules, including those blocks generated by the compiler. The functions of the blocks and the way in which they are used by the library are described in Chapter 3. In the diagrams, all offsets are in hexadecimal.

The appendix includes an example of the chaining of I/O control blocks.

```
     0       7 8     15 16      23 24      31
    ┌───────────────────┬───────────────────┐
  0 │       DPRO        │       DCLA        │
    ├───────────────────┼───────────────────┤
  4 │       DBLK        │       DLRL        │
    ├─────────┬─────────┼─────────┬─────────┤
  8 │  DCLD   │  DBNO   │  DCLB   │  DCLC   │
    ├─────────┴─────────┼─────────┴─────────┤
  C │       DXAL        │    (Reserved)     │
    ├───────────────────┴───────────────────┤
 10 │              (Reserved)               │
    ├───────────────────────────────────────┤
 14 │              (Reserved)               │
    ├─────────┬─────────────────────────────┤
 18 │  DFLN   │                             │
    ├─────────┘                             │
    │                                       │
    │                 DFIL                  │
    │                                       │
    │                                       │
    │                                       │
    └───────────────────────────────────────┘
```

Figure 60. Format of the Declare Control Block (DCLCB)

DPRO: Halfword binary integer (set by the linkage editor) specifying the offset, within the pseudo-register vector (PRV), of the pseudo-register associated with the declared file.

DCLA: Four four-bit codes specifying the file type, organization, access and mode:

Byte 1

	Type
0001 xxxx	STREAM
0010 xxxx	RECORD

	Organization
xxxx 0000	CONSECUTIVE
xxxx 0001	INDEXED
xxxx 0010	REGIONAL (1)
xxxx 0011	REGIONAL (2)
xxxx 0100	REGIONAL (3)

(Stream-oriented I/O is supported only for data sets of CONSECUTIVE organization.)

Byte 2

	Access
0001 xxxx	SEQUENTIAL
0010 xxxx	DIRECT

(These are used for record-oriented I/O only.)

	Mode
xxxx 0001	INPUT
xxxx 0010	OUTPUT
xxxx 0100	UPDATE
xxxx 1000	BACKWARDS

(Stream-oriented I/O uses INPUT and OUTPUT only.)

DBLK: Halfword binary integer specifying the length, in bytes, of the blocks within the data set:

F-format records: block length specified for data set (constant for all blocks except possibly the last one).

U-, V-, VS- or VBS-format records: maximum length of any block in data set.

DLRL: Halfword binary integer specifying the length, in bytes, of the records within the data set. Two or more records may be grouped (blocked) to form one physical block.

F-format records: record length specified for data set (constant for all records).

V-, VS- or VBS-format records: maximum length of any record in the data set.

U-format records: this specification is not permitted; the block size defines the record length.

DCLD: One byte containing ENVIRONMENT options:

Bit	Option
0	LEAVE
1	COBOL file
2	CTLASA
3	CTL360
4	INDEXAREA
5	NOWRITE
6	REWIND
7	GENKEY

DBNO: One-byte binary integer specifying the number of buffers to be allocated to the file when it is opened, as specified by the BUFFERS option.

DCLB: One byte containing attribute codes:

Bit	Attribute
0	KEYED
1	EXCLUSIVE
2	BUFFERED
3	UNBUFFERED
4	(Reserved)
5	(Reserved)
6	(Reserved)
7	(Reserved)

DCLC: Eight-bit code which specifies the format of records within the data set:

Bits	Code	Format
0 and 1	01	V
0 and 1	10	F
0 and 1	11	U
2	-	(Reserved)
3	1	Blocked
4	1	VS/VBS
5	1	PRINT
6	-	(Reserved)
7	-	(Reserved)

DXAL: Halfword binary integer specifying the count in the INDEXAREA area environment option.

DFLN: One-byte binary integer specifying the length (minus one) in bytes of the file name in the following field.

DFIL: Character string, up to 31 bytes long, specifying the name of the file. If there is no TITLE option in the OPEN statement, the first eight characters of this name are used as the name of the DD statement associated with the file during program execution. (The compiler will have padded the name with blanks to extend it to at least eight characters in length.)

EVENT VARIABLE

```
     0     7 8         15 16                  31
    ┌───────┬──────────────────────────────────┐
  0 │ EVF1  │             EVEC                 │
    ├───────┼──────────────────────────────────┤
  4 │ EVF2  │             EVIO                 │
    ├───────┴──────────────────────────────────┤
  8 │                     EVCF                 │
    ├──────────────────────────────────────────┤
  C │                     EVCB                 │
    ├───────────────────┬──────────────────────┤
 10 │       EVST        │      Reserved        │
    ├───────────────────┴──────────────────────┤
 14 │                     EVFF                 │
    ├──────────────────────────────────────────┤
 18 │                     EVFB                 │
    ├──────────────────────────────────────────┤
 1C │                     EVPR                 │
    └──────────────────────────────────────────┘
```

Figure 61. Format of the Event Variable

In a multitasking environment, event variables are placed in two chains:

1. The file chain, which is anchored in the TEVT field of the FCB and includes all active event variables related to a file and for which there is no corresponding IOCB. This chain enables all associated event variables that are not being waited on to be set inactive, complete, and abnormal when a file is closed.

2. The task chain, which is anchored in the pseudo-register IHEQEVT, and includes all active I/O event variables associated with the task. This chain facilitates the setting of those event variables that are not being waited on inactive, complete, and abnormal on termination of the task.

An example of the chaining of event variables is given at the end of this appendix.

EVF1: 8-bit code containing implementation flags:

Flags	Code	Name
Active event variable	1000 0000	EMAC
I/O associations	0100 0000	EMIO
No WAIT required	0010 0000	EMNW
FCB address contained in EVEC	0001 0000	EMFC
This event variable is to be checked	0000 1000	EMCH
DISPLAY event variable	0000 0100	ENDS
IGNORE option with this event	0000 0010	EMIG

EVEC: Contains the address of the DECB associated with the event, or the address of the FCB when no IOCB was obtained, e.g., when READ IGNORE(0) is executed.

EVF2: PL/I ECB flag byte:

Flags	Code	Name
Wait	1000 0000	EMWB
Complete	0100 0000	EMCP

EVIO: Not used.

EVCF: Event variable chain-forward pointer (task).

EVCB: Event variable chain-back pointer (task).

EVST: Status field:
Normal status value: All zeros.
Abnormal status value: Low-order bit is 1, remainder is zero (unless set otherwise by STATUS pseudo-variable).

EVFF: Event variable chain-forward pointer (file).

EVFB: Event variable chain-back pointer (file).

EVPR: Address of the PRV of the task in which the associated I/O event was initiated.

```
     0     7 8        15 16          31
    ┌─────────────────────────────────┐
  0 │              XCFF               │
    ├─────────────────────────────────┤
  4 │              XCBF               │
    ├─────────────────────────────────┤
  8 │              XCFT               │
    ├─────────────────────────────────┤
  C │              XCBT               │
    ├─────────────────────────────────┤
 10 │              XPRV               │
    ├──────┬──────────┬───────────────┤
 14 │ XFL1 │(Reserved)│     XSTC      │
    ├──────┴──────────┴───────────────┤
 18 │                                 │
    │              XQNM               │
    │                                 │
    ├──────┬──────────────────────────┤ ──────
 20 │ XLRN │      XKYI/XREG           │   Λ
    ├──────┴──────────────────────────┤   │
 24 │                                 │   │
    │                                 │   │
    │              XKYR               │ XRNM
    │                                 │   │
    │                                 │   │
    │                                 │   │
    │                                 │   V
    └─────────────────────────────────┘ ──────
```

Figure 62. Format of Exclusive Block

Exclusive blocks are placed in two chains:

1. The task chain, which is anchored in the pseudo-register IHEQXLV, and enables all records locked in a task to be unlocked when the task is terminated.

2. The file chain, which is anchored in the TXLV field of the FCB, and facilitates the freeing of all exclusive blocks related to the file when it is closed, and facilitates a check on whether a record is already locked.

An example of the chaining of exclusive blocks is given at the end of this appendix.

XCFF:	Chain-forward pointer (file).
XCBF:	Chain-back pointer (file).
XCFT:	Chain-forward pointer (task).
XCBT:	Chain-back pointer (task).
XPRV:	Address of the PRV for the task in which the exclusive block was created.
XFL1:	Flags: XLOK: Code 1000 0000 indicates that the record associated with the exclusive block is locked owing to a READ operation or an incomplete REWRITE or DELETE operation.
XSTC:	Lock statement count: the number of incomplete I/O operations that currently refer to the exclusive block.
XQNM:	Eight-byte qname used in the ENQ and DEQ macro instructions. The first word contains the address of the FCB, right-aligned, and the second contains zero.
XRNM:	The rname used in the ENQ and DEQ macro instructions:

XLRN: One byte containing the length of the rname.

XKYI/XREG:

XKYI:	INDEXED files (unblocked records): Key of record being locked. INDEXED files (blocked records): A(FCB).
XREG:	REGIONAL files: Region number of the record being locked. (This field may extend beyond byte 23.)

XKYR: REGIONAL(2) and (3) files: The recorded key of the record being locked.

	0–7	8–15	16–23	24–31
-8	TVAL			
-4	TRES			
0	TFLX	TDCB		
4	TTYP	TACM		
8	TFLA	TFLB	TLEN	
C	TFIO	TDCL		
10	TCBA			
14	TREM		TMAX	
18	TREC			
1C	TCNT			
20	TPGZ		TLNZ	
24	TLNN		TFLC	TFLD
28	TFLE	TFOP		
2C	TFLF	TTAB		
30	DCB			

●Figure 63. FCB for Stream-Oriented I/O

TVAL: Word containing bits indicating which statements are valid for this file

TRES: Reserved

TFLX: Eight-bit code specifying error and exceptional conditions:

Conditions	Code	Name
EOF on data set	1000 0000	TMEF
Error on output	0100 0000	TMOE
Error on input	0010 0000	TMIE
Error on data field	0001 0000	TMIT
Do not raise TRANSMIT	0000 1000	TMNX
List terminator	0000 0010	TMLC
ENDPAGE raised	0000 0001	TMEP

TDCB: Address of the DCB part of the FCB.

	0–7	8–15	16–23	24–31
-8	TVAL			
-4	TRES			
0	TFLX	TDCB		
4	TTYP	TACM		
8	TFLA	TFLB	TLEN	
C	TFIO	TDCL		
10	TLAB/TCBA			
14	TPKA			
18	TBBZ/TREL			
1C	TADC			
20	TLRR			
24	TLRL		TFLC	TFLD
28	TFLE	TFOP		
2C	TFLF	TFMP	(Reserved)	
30	TEVT			
34	Zero			
38	TXLV*			
3C	Zero*			
40	TXLZ*			
44	DCB			

* These fields are omitted in non-multitasking environment: DCB commences at byte 38.

●Figure 64. FCB for Record-Oriented I/O

TTYP: Eight-bit code specifying I/O type:

Type	Code	Name
STREAM I/O	xxxx 0000	TMDS
RECORD I/O	xxxx 0001	TMRC
STRING I/O	xxxx 0010	TMST
Temporary flags,	1000 xxxx	TMT1
valid for single	0100 xxxx	TMT2
I/O call only	0010 xxxx	TMT3
	0001 xxxx	TMT4

TACM: Address of I/O transmit module, which interfaces with data management access methods. The names of all such library modules are IHEIT*, where * is a letter identifying the module.

TFLA: Two four-bit codes specifying the record format and the current file mode:

Format	Code	Name
V (variable)	0001 xxxx	TMVB
F (fixed)	0010 xxxx	TMFX
U (undefined)	0100 xxxx	TMUN
ASA control/print file	1xxx xxxx	TMAS

Mode	Code	Name
INPUT	xxxx 0001	TMIN
OUTPUT	xxxx 0010	TMOP
UPDATE	xxxx 0100	TMUP
BACKWARDS	xxxx 1000	TMBK

TFLB: Eight-bit code specifying the file attributes:

Attribute	Code	Name
EXCLUSIVE	1xxx xxxx	TMEX
UNBUFFERED	x1xx xxxx	TMBU
Hidden buffers	xx1x xxxx	TMHB
SYSPRINT file	xxx1 xxxx	TMPT
Hidden buffer may be required	xxxx x1xx	TMHQ
KEYED	xxxx xx1x	TMKD
DIRECT	xxxx xxx1	TMDR

TLEN: Halfword binary integer, specifying the length, in bytes, of the FCB.

TFIO: Eight-bit code specifying the type of I/O operation:

Operation	Code	Name
PUT	1000 0000	TMPW
GET	0100 0000	TMGR
EVENT option with IGNORE option	0000 0010	TMEI
COPY option	0000 0001	TMCY

TDCL: Address of the DCLCB defining the file.

TCBA/TLAB:

STREAM: TCBA: Address of next available byte in a buffer.

RECORD: TLAB:
Sequential: Address of last IOCB obtained.
Direct: Address of first IOCB in chain.
TCBA:
Sequential: Address of last record located.

TREM/TMAX/TPKA:

STREAM: TREM: Number of bytes remaining in current record. This value is equal to TLNZ when the record is initialized for output.
TMAX: Halfword binary integer specifying the number of bytes in a record:

Input: the number of bytes read.

Output: the number of bytes initially available.

For V format records, this number includes the four-byte record control field; for all record formats, it includes the ASA control byte (if present).

RECORD: TPKA: Address of previous key. (Used for SEQUENTIAL access to REGIONAL data sets, LOCATE creation of INDEXED data sets, and padding key for SEQUENTIAL INDEXED data sets.)

TREC/TBBZ/TREL:

STREAM: TREC: Address of buffer workspace (paper-tape input, U-format output).

RECORD: TBBZ: Length of IOCB. The first byte contains the subpool number.
TREL: Relative record count. (Used only for SEQUENTIAL access to REGIONAL data sets.)

TCNT/TADC:

STREAM: TCNT: Fullword binary integer specifying the number of scalar items transmitted during the most recent I/O operation (GET or PUT) on the file.

RECORD: TADC: Address of the adcon list.

TPGZ/TLNZ/TLRR:

STREAM: TPGZ: Halfword binary integer specifying the maximum number of lines per page. This field is only used for PRINT files. A default value of 60 lines is assumed if:

1. the OPEN statement that opens the file does not include the PAGESIZE option, or

2. an implicit open occurs.

TLNZ: Halfword binary integer specifying the maximum number of characters per line. A default line size is obtained from the record length specified in the ENVIRONMENT attribute if:

1. the OPEN statement that opens the file does not include the LINESIZE option, or

2. an implicit open occurs.

If the ENVIRONMENT attribute is not specified, the record length used is that specified in the associated DD statement.

If none of these specifies a record size, and if the file is a print file, a default length of 120 characters per line is assumed.

The TLNZ value includes all characters available within a line.

RECORD: TLRR: Address of IOCB of last complete READ operation. This is required whenever the EVENT option is used; it provides a means of identifying the last complete READ operation when a REWRITE is executed. In the case of spanned records this field holds the length of the previously read record if the previous operation was a READ SET.

TLNN/TLRL:

STREAM: TLNN: Halfword binary integer specifying the current line number.

RECORD: TLRL: Maximum logical record length for the file.

TFLC: Two 4-bit codes giving:

1. Type of device.

2. Further file history.

Device	Code	Name
Paper tape	1000 0000	TMPA
Printer	0100 0000	TMPR
Previous operation was READ with SET option	0000 1000	TMPS
Attempt to close in wrong task	0000 0100	TMDT
OPEN or CLOSE in progress	0000 0010	TMOC

TFLD: Eight-bit code specifying the organization of the data set associated with the file:

Organization	Code	Name
CONSECUTIVE	X'00'	TMCN
INDEXED	X'04'	TMIX
REGIONAL (1)	X'08'	TMR1
REGIONAL (2)	X'0C'	TMR2
REGIONAL (3)	X'10'	TMR3

TFLE: Eight-bit code specifying the history of the file:

History	Code	Name
Preceding operation a READ	1000 0000	TMRP
IGNORE in progress	0100 0000	TMIG
CLOSE in progress	0010 0000	TMCL
End of the extent reached by the last operation	0001 0000	TMET
Preceding operation a REWRITE	0000 1000	TMWP
Preceding operation a LOCATE	0000 0100	TMLT
I/O condition on CLOSE	0000 0010	TMCC
Implicit CLOSE	0000 0001	TMCT

TFOP: Address of the prior FCB opened in the current task, or zero (if FCB is the first FCB opened).

TFLF: Eight-bit code specifying the load module code (used by IHECLS, IHECLT and IHECTT to specify module names in the DELETE macro):

STREAM:

Miscellaneous	Code	Name
TAB table exists	0000 0001	TMTB

RECORD:

Module Code	Code	Name
QSAM	X'00'	TMQS
BDAM	X'04'	TMBD
QISAM	X'08'	TMQI
BISAM	X'0C'	TMBI
BSAM	X'10'	TMBS
BSAM load mode	X'14'	TMBL
Tab control table exists	X'01'	TMTB

TTAB: Address of TAB control table (PRINT files only).

TFMP: RECORD I/O only. This flag is used by exclusive files to act as a lockout flag when updating the chains of IOCBs and exclusive blocks. A TS loop is performed on this byte until it is freed. When the chaining operation is complete, the byte is set to zero.

TEVT: Pointer to chain of active I/O event variables associated with the file, but for which there is no corresponding IOCB: enables the event variables to be set complete, inactive, and abnormal when the file is closed.

TXLV: Pointer to chain of exclusive blocks associated with locked records of the file: enables locked records to be unlocked when the file is closed. (Used only in a multitasking environment.)

TXLZ: Length of exclusive block: the first byte contains X'01'(the number of the subpool in which storage for the block is allocated).

DCB: This field, variable in length and format, is the data control block defined by data management for the various access methods.

```
      0       7 8        15 16                 31
     ┌─────────┬──────────────────────────────────┐─────────────────────
   0 │  BACT   │              BPIO                │           Λ
     ├─────────┴──────────────────────────────────┤           |
   4 │                        BNIO                │           |
     ├─────────┬──────────────────────────────────┤           |
   8 │  BERR   │              BFCB                │           |
     ├─────────┴──────────────────────────────────┤           |
   C │                        BREQ                │           |
     ├───────────────────────┬────────────────────┤           |
     │BERC/BEFC/BXTC/BKYC    │        BRCC        │          IOCB
     ├───────────────────────┴────────────────────┤       foundation
  14 │                        BRVS                │           |
     ├────────────────────────────────────────────┤           |
  18 │                        BEVN                │           |
     ├────────────────────────────────────────────┤           |
  1C │                     BDF1/BBF1              │           |
     ├───────────────────────┬────────────────────┤           |
  20 │       BDF2/BBF2       │ BDF3/(Reserved)    │           |
     ├───────────────────────┴────────────────────┤           |
  24 │                     BDF4/BBF3              │           |
     ├────────────────────────────────────────────┤           |
  28 │                BDF5/BBF3(contd.)           │           V
     ├────────────────────────────────────────────┼─────────────────────
  2C │                     BECB/BEXD              │     Λ        Λ
     ├───────────────────────┬────────────────────┤     |        |
  30 │         BTYP          │        BLEN        │   BSAM   BDAM/BISAM
     ├───────────────────────┴────────────────────┤   DECB      DECB
  34 │                        BDCB                │     |         |
     ├────────────────────────────────────────────┤     |         |
  38 │                        BARE                │     |         |
     ├────────────────────────────────────────────┤     |         |
  3C │                     BSTS/BLOG              │     V         |
     ├────────────────────────────────────────────┼──────         |
  40 │                     BKVS/BKEY              │               |
     ├────────────────────────────────────────────┤               |
  44 │                     BBLK/BEXI              │               V
     ├────────────────────────────────────────────┼─────────────────────
  48 │                     BDBF/BXLV              │               Λ
     ├────────────────────────────────────────────┤               |
  4C │                     (Reserved)             │               |
     ├────────────────────────────────────────────┤           BDAM/BSAM
  50 │                                            │             Hidden
   . │                        BBBF                │             buffer
   . │                                            │              area
   . │                                            │               |
   . │                                            │               |
     │                                            │               V
     └────────────────────────────────────────────┘─────────────────────
```

<u>Note:</u> (The IOCB includes the Data Event Control Block (DECB) for the BSAM and BDAM/BISAM Interfaces)

Figure 65. Format of the I/O Control Block (IOCB)

BACT: One byte containing an activity flag (used only in direct access):

Code	Meaning
X'FF'	In use
X'00'	Free

BPIO: Chain-back address of the previous I/O control block.

BNIO: Chain-forward address of the next I/O control block.

BERR: Flag byte for record-oriented I/O situations:

Situation	Code	Name
IOCB has been checked	0000 0001	BMCH
I/O error exists	0000 0010	BMER
End-of-file has occurred	0000 0100	BMEF
Possible lock for REWRITE	0000 1000	BMPR
Lock for REWRITE	0001 0000	BMNR
IOCB for BISAM READ UPDATE mode	0100 0000	BMDF
Dummy buffer acquired	1000 0000	BMDB

BFCB: Address of the FCB for the file.

BREQ: Request control block. Four-byte field specifying the request codes for associated operations (as passed by the compiled calling sequence):

Byte 1	Operation
X'00'	READ
X'04'	WRITE
X'08'	REWRITE
X'0C'	DELETE
X'10'	LOCATE
X'14'	UNLOCK
X'18'	WAIT

Byte 2	Option Set 1
X'00'	None/SET
X'04'	IGNORE
X'08'	INTO/FROM

Byte 3	Option Set 2
X'00'	None
X'04'	KEYTO
X'08'	NOLOCK

Byte 4	Option Set 3
X'40'	VARYING record variable (INTO)
X'80'	VARYING KEYTO

BERC/BEFC/BXTC/BKYC: Error codes for various conditions.

BERC: ERROR condition

BEFC: ENDFILE condition

BXTC: TRANSMIT condition

BKYC: KEY condition

(See Chapter 6 for details of these codes.)

BRCC: Error code for RECORD condition. (See Chapter 6 for details of these codes.)

BRVS: Address of RDV or SDV for record variable.

BEVN: Address of event variable; zero, if none exists for associated operation.

BDF1/BBF1:

BSAM: BDF1: Address of the user's record variable.

BDAM: BBF1: Address of the user's record variable.

BDF2/BBF2:

BSAM: BDF2: Length, in bytes, of the user's record variable.

BDAM: BBF2: Length, in bytes, of the user's record variable.

BDF3:

BSAM: Length, in bytes, of the KEYTO area.

BDAM: (Reserved)

BDF4/BBF3:

BSAM: BDF4: Address of the KEYTO area.

BDAM: BBF3: Relative record or track number (BLKREF).

BDF5: BSAM: Relative record number (REGIONAL (1)).

BECB/BEXD:

BECB: The data management event control block (ECB).

BEXD: If BDAM is used, bytes 2 and 3 (= BEXD) of this field contain the BDAM exception codes. For definitions of these codes, see <u>IBM System/360 Operating System: Supervisor and Data Management Macro Instructions</u>.

BTYP: Type of I/O operation (set by data management macro).

BLEN: Length, in bytes, of the records to be transmitted.

BDCB: Address of the DCB.

BARE:

Hidden buffers:	Address of the appended buffer.
No hidden buffers:	Address of the record variable.

BSTS/BLOG:

BSAM: BSTS: Address of the status indicator.

BDAM: BLOG: Address of the IOB (I/O block; see IBM System/360 Operating System: System Programmer's Guide.

BISAM: BLOG: Address of the logical record.

BKVS/BKEY

BSAM: BKVS: Address of SDV for KEYTO.

BDAM: BKEY: Address of KEY

BBLK/BEXI:

BBLK: Address of BLKREF, the relative record or track number (i.e., the address of BBF3).

BEXI: If BISAM is used, one byte (= BEXI) contains the BISAM exception codes. For definitions of these codes, see IBM System/360 Operating System: Supervisor and Data Management Macro Instructions.

BDBF/BXLV:

BSAM and BISAM: BDBF: Start of hidden buffer.

BDAM: BXLV: Address of the exclusive block (if any) associated with record being referenced.

BBBF: Start of BDAM/BISAM hidden buffer.

	SEQUENTIAL				DIRECT			
	CONSECUTIVE	REGIONAL (KEYED)			REGIONAL			INDEXED
		(1)	(2)	(3)	(1)	(2)	(3)	
F-format records	A B	A B 8	A B D_1 D_2	A B D_1 D_2	A C 8	A C D_1	A C D_1	A C D_1 D_2 16 (Note 1)
V-format records	A B D_2	-	-	A B D_1 D_2	-	-	A C D_1 D_2	-
U-format records	A B	-	-	A B D_1 D_2	-	-	A C D_1	-

A: Size of IOCB foundation
B: Size of BSAM DECB
C: Size of BDAM/BISAM DECB
D: Size of hidden buffer:
D_1: Length of recorded key
D_2: Length of block (record)

Note 1: If RKP ≠ 0, then $D_1 = 0$. If RKP = 0 then for blocked records: $D_1 = L$, and for unblocked records: $D_1 = 2L$, where L = length of recorded key.
Note 2: The data value is obtained by summing the sizes given under each entry.

Figure 66. Values used in computing size of IOCB for various access methods

OPEN CONTROL BLOCK (OCB)

```
0         4          8          12
┌─────────┬──────────┬──────────┬─────────┐
│  Type   │    0     │  Access  │  Mode   │
└─────────┴──────────┴──────────┴─────────┘

16        20         24         28      31
┌─────────┬──────────┬──────────┬─────────┐
│ Flag A  │  Flag B  │  Flag C  │ Flag D  │
└─────────┴──────────┴──────────┴─────────┘
```

Figure 67. Format of the Open Control Block (OCB)

Type	STREAM		0001
	RECORD		0010
Access	SEQUENTIAL		0001
	DIRECT		0010
Mode	INPUT		0001
	OUTPUT		0010
	UPDATE		0100
	BACKWARDS		1000
Flag A		Bit: 0	KEYED
		1	EXCLUSIVE
		2	BUFFERED
		3	UNBUFFERED
Flags B & C			(Reserved)
Flag D		Bit: 0	(Reserved)
		1	PRINT
		2	(Reserved)
		3	(Reserved)

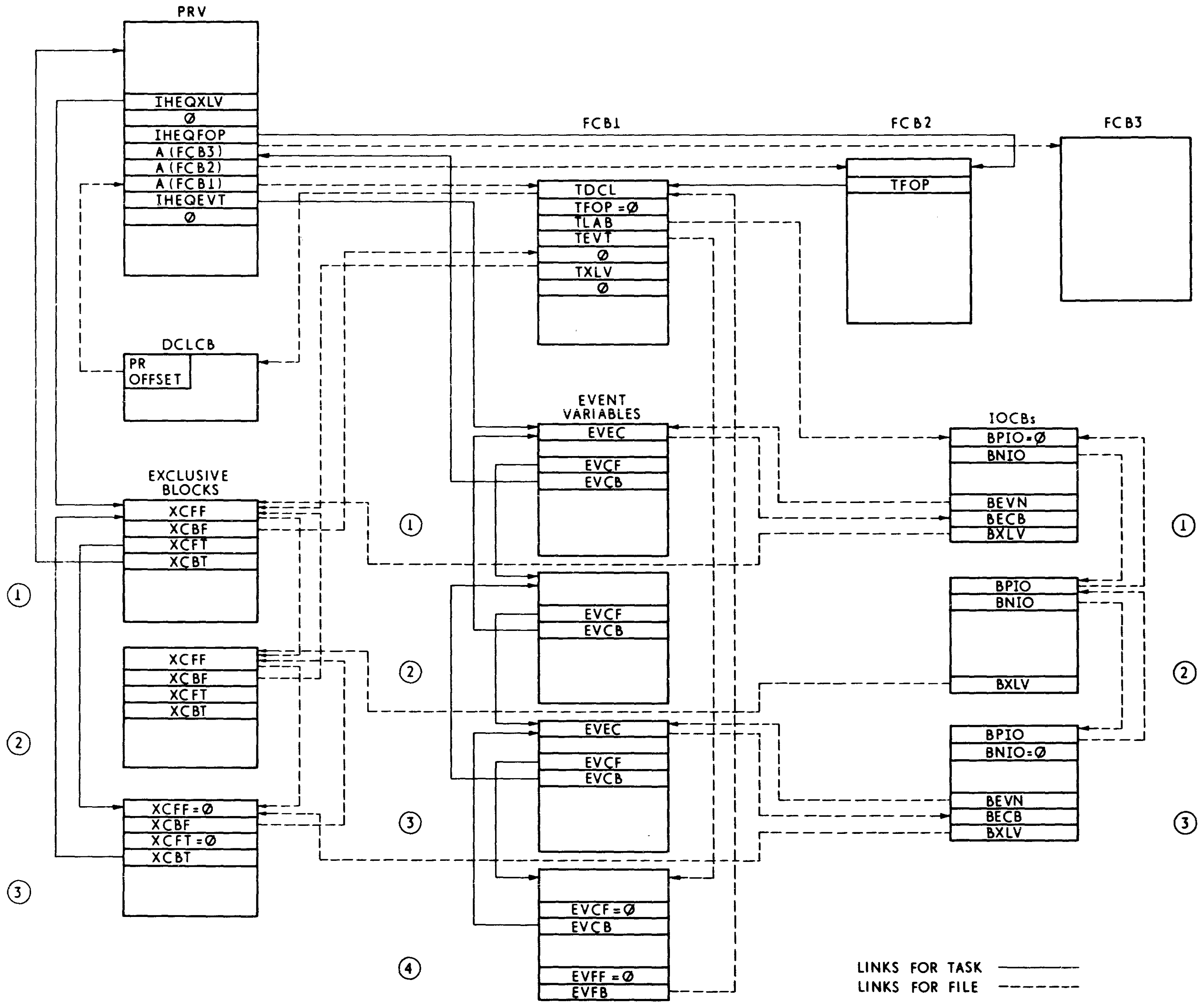


Figure 68. Example of Chaining of I/O Control Blocks

EXAMPLE OF CHAINING

Figure 68 contains an example of the chaining of FCBs, IOCBs, event variables, and exclusive blocks in a single task.

Files

The task has opened two files, and the addresses of their FCBs (FCB1 and FCB2) are stored in the PRV; the FCBs are placed in a chain that is anchored in the pseudo-register IHEQFOP and uses the TFOP fields in the FCBs. The task also has access to another file that was opened in a higher task; the address of the FCB for this file (FCB3) was copied into the PRV when the task was attached. (Note that this FCB does not appear in the IHEQFOP chain.) A DCLCB exists for each file declared, but only the one corresponding to FCB1 is shown in Figure 68; this file is an exclusive file that has been opened for DIRECT UPDATE.

IOCBs

Three of the current I/O operations that refer to FCB1 required IOCBs. The IOCBs are placed in a chain anchored in the TLAB field of the FCB so that they can be freed when the file is closed. The BXLV field in each IOCB addresses the corresponding exclusive block. The EVENT option was used with two of the I/O operations: the BEVN fields in IOCBs 1 and 3 therefore point to the corresponding event variables. (The third operation originated in another task.)

Event Variables

The task has four active I/O event variables. These are chained from the pseudo-register IHEQEVT so that, on termination of the task, they can be set complete, inactive, and abnormal. (Note that the address in the chain-back field EVCB in event variable 1 is not that of IHEQEVT, but that of the field three words higher: IHEQEVT is thus in the same position relative to this address as EVCB is relative to the first byte of the event variable.) Event variables 1, 3, and 4 relate to the file corresponding to FCB1, and must be set complete, inactive, and abnormal when the file is closed. Communication with event variables 1 and 3 is established via the corresponding IOCBs. But event variable 4, which relates to an I/O operation for which an IOCB was not required, is placed in a chain anchored in the TEVT field of the FCB. Event variable 2 is related to an I/O operation on another file in the task.

Exclusive Blocks

For REGIONAL files and INDEXED files with unblocked records, an exclusive block exists for each record currently locked; all those shown refer to the file corresponding to FCB1. (If the files have blocked records, only one exclusive block exists for each file in each task; it is created the first time a record in the file is locked, and is not freed until the file is closed.) The exclusive blocks are placed in a chain anchored in the TXLV field of the FCB so that the blocks can be freed when the file is closed. Only two of the records have been locked by this task, and their exclusive blocks (1 and 3) are placed in a chain anchored in pseudo-register IHEQXLV so that the records can be unlocked on termination of the task. (Note that the chain-back fields, XCBT and XCBF, in exclusive block 1 point, not to IHEQXLV and TXLV, but to fields in the PRV and FCB1 that have the same positions relative to IHEQXLV and TXLV as the start of the exclusive block has relative to XCBT and XCBF.)

APPENDIX J: STORAGE-MANAGEMENT CONTROL BLOCKS

This appendix gives the formats of the control blocks used by the non-multitasking storage-management modules of the PL/I Library; the formats of the multitasking equivalents are given in Appendix K. The functions of the blocks and the way they are used are described in Chapter 4. In the diagrams, all offsets are in hexadecimal.

AREA VARIABLE

```
     0       7 8                       31
    ┌─────────┬─────────────────────────┐
  0 │See Note │  Length of Area Variable │
    ├─────────┴─────────────────────────┤
  4 │      Offset of End of Extent      │
    ├───────────────────────────────────┤
  8 │   Offset of Largest Free Element  │
    ├───────────────────────────────────┤
  C │              See Note             │
    ├───────────────────────────────────┤
    │                                   │
    │                                   │
    │                                   │
    │                                   │
    │                                   │
    │                                   │
    │                                   │
    │                                   │
    └───────────────────────────────────┘
```

Note: If the area variable contains a free list, bit 0 of the first byte is set to 1, and the fourth word is set to 0.

Figure 69. Format of Area Variable

	0 7	8 31
0	Flags	Length
4	Chain-back address	
8	Chain-forward address	
C . . . 44	Register save area	
48	Current file	
50	Invocation count	
58	OPTIONAL ENTRIES: Display Statement number ON fields Dope vectors AUTOMATIC data Workspace Parameter lists	

Figure 70. Format of the Dynamic Storage Area (DSA)

Bit	Meaning = 0	Meaning = 1
0	Always = 1	
1	No statement number field in DSA	Statement number field in DSA
2	No dummy ON field for STRINGRANGE	STRINGRANGE field created as for other ON conditions
3	Procedure DSA	Begin block DSA
4	No dummy ON field for SUBSCRIPTRANGE	SUBSCRIPTRANGE field created as for other ON conditions
5	Non-recursive DSA, without display update field	Recursive DSA, with display update field
6	No ON fields	ON fields
7	No dummy ON field for SIZE	SIZE field created as for other ON conditions

Figure 71. Format of the DSA flag byte

The minimum size of a non-multitasking DSA is X'64' bytes.

Standard Entries

Standard Save Area: The area starting with the flags and continuing up to and including the register save area. (See Figure 55 and associated text.)

Current File: This field is eight bytes long; its use is described in 'Current File' in Chapter 3. In a multitasking environment, the first byte is used as the SYSPRINT resource counter; see 'SYSPRINT in Multitasking' in Chapter 3.

Invocation Count: This field is eight bytes long and contains:

1st word: Environment chain-back address or zero

2nd word: Invocation count

Optional Entries

Display: This field is eight bytes long and contains:

1st word: Pseudo-register offset

2nd word: Pseudo-register update

If it occurs at all, the display field always appears at offset 58.

Statement Number: This field is four bytes long; it is described in 'Error and Interrupt Handling'. If it occurs at all, the statement number always appears at offset 60; bytes 60-63 are always set to zero. If there is no statement number, this field can be used for optional DSA entries, e.g., ON fields.

ON fields: Each ON field is two words long. The ON fields are described in 'ON Conditions' under 'Error and Interrupt

Handling'. The position of the first ON field depends on whether there are entries in the display update and statement number fields:

1. No display update, no statement number: ON fields begin at offset 58.
2. Display update, but no statement number: ON fields begin at offset 60.
3. Statement number (with or without a display update): ON fields begin at offset 64.

The last ON field is indicated by bit 0 = 1 in the second word.

Remaining Entries

The dope vector formats are described in Appendix H ('Compiler-Generated Control Blocks'). The AUTOMATIC data, workspace and parameter lists areas are provided for use by the compiler.

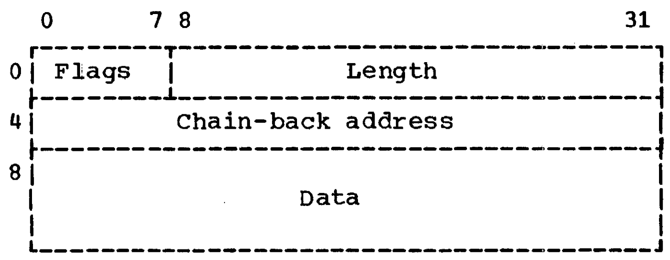
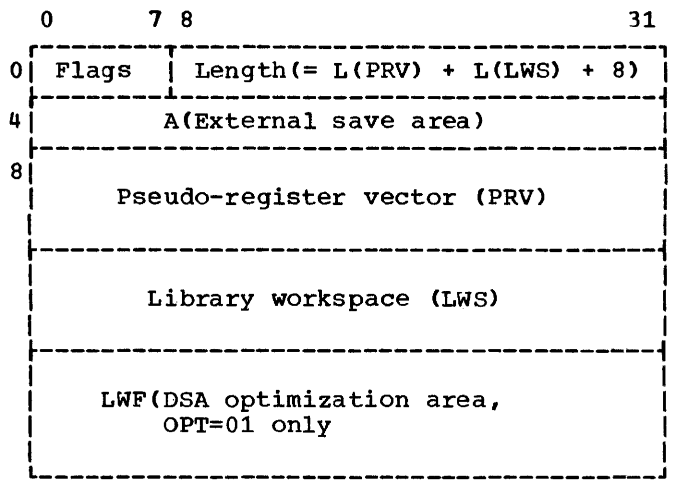


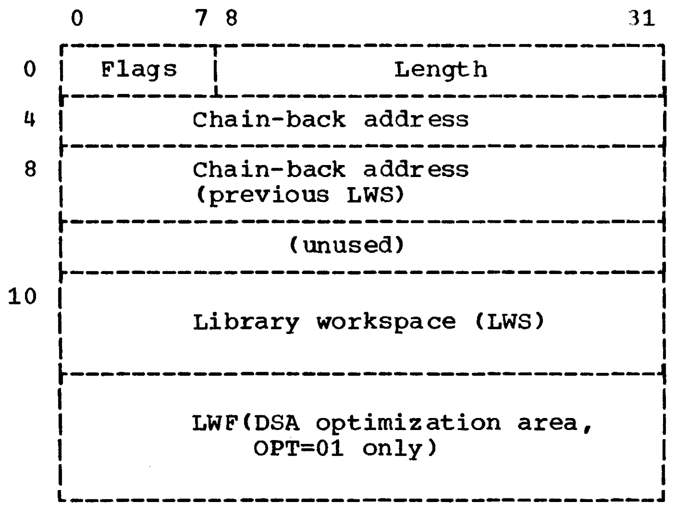
Figure 72. Format of the Variable Data Area (VDA)


0
7 8
31
0
Flags
Length(= L(PRV) + L(LWS) + 8)
4
A(External save area)
8
Pseudo-register vector (PRV)
Library workspace (LWS)
LWF(DSA optimization area, OPT=01 only

● Figure 74. Format of the PRV VDA

Bit		Meaning
0 1 2 3	4 5 6 7	
0 0 1 0	0 0 0 0	Ordinary VDA
0 0 1 0	0 0 0 1	VDA obtained for a library subroutine
0 0 1 0	0 1 0 1	VDA containing a secondary LWS
0 0 1 0	1 0 0 1	PRV VDA

Figure 73. Format of the VDA flag byte


0
7 8
31
0
Flags
Length
4
Chain-back address
8
Chain-back address (previous LWS)
(unused)
10
Library workspace (LWS)
LWF(DSA optimization area, OPT=01 only)

● Figure 75. Format of LWS VDA

APPENDIX K: MULTITASKING CONTROL BLOCKS

This appendix describes the control blocks used by the multitasking storage-management modules of the PL/I Library. The way in which they are used by the library is described in Chapter 5. In the diagrams, all offsets are in hexadecimal.

```
     0      7 8                          31
     ┌───────┬────────────────────────────┐
   0 │ Flags │          Length            │
     ├───────┴────────────────────────────┤
   4 │        Chain-back address          │
     ├────────────────────────────────────┤
   8 │        Chain-forward address       │
     ├────────────────────────────────────┤
   C │                                    │
   . │                                    │
   . │        Register save area          │
   . │                                    │
  44 │                                    │
     ├────────────────────────────────────┤
  48 │                                    │
     │                                    │
     │        Current file                │
     │                                    │
     ├────────────────────────────────────┤
  50 │                                    │
     │                                    │
     │        Invocation count            │
     │                                    │
     ├────────────────────────────────────┤
  58 │                                    │
     │        Display                     │
     │                                    │
     ├───────┬────────────────────────────┤
  60 │ Flags │      Statement number      │
     ├───────┴────────────────────────────┤
  64 │        A(Task variable chain)      │
     ├────────────────────────────────────┤
  68 │                Zero                │
     ├────────────────────────────────────┤
  6C │        ON fields                   │
     │        Dope vectors                │
     │        AUTOMATIC data              │
     │        Workspace                   │
     │        Parameter lists             │
     └────────────────────────────────────┘
```

Figure 76. Format of the Dynamic Storage Area (DSA) for Multitasking

The minimum size of a multitasking DSA is X'6C' bytes.

The multitasking DSA contains two fields that do not appear in the non-multitasking DSA (Appendix J): the fullword commencing at byte 64 contains the address of the first task variable in the task-variable chain (if any); the following fullword is always set to zero. The presence of a task variable chain is indicated by bit 0 = 1 in byte 60. The Get DSA routine IHETSAD differs from its non-multitasking equivalent only in that it sets the doubleword commencing at byte 64 to zero.

EVENT VARIABLE

```
     0       7 8      15 16    23 24     31
    ┌─────────┬──────────────────────────────┐
 0  │ Flags   │            Zero              │
    ├─────────┴──────────────────────────────┤
 4  │                ECB                     │
    ├────────────────────────────────────────┤
 8  │              Reserved                  │
    ├────────────────────────────────────────┤
 C  │              Reserved                  │
    ├───────────────────┬────────────────────┤
10  │      Status       │ Statement Number   │
    ├─────────┬─────────┼─────────┬──────────┤
14  │Reserved │   MCF   │   WTF   │Reserved  │
    ├─────────┴─────────┴─────────┴──────────┤
18  │          Infinite Wait ECB             │
    ├────────────────────────────────────────┤
1C  │          Wait to Terminate ECB         │
    └────────────────────────────────────────┘
```

● Figure 77. Format of the Event Variable

The task event variable is not chained.

<u>Flags</u>:

<u>Flag</u>	<u>Code</u>
Active event variable	1000 0000
Multitasking (non-I/O) event variable	0000 0000
Normal PL/I termination	0010 0000
Abnormal PL/I termination	0001 0000
Event variable being waited on	0000 0001

<u>ECB</u>: This is the control program event control block. Bit 0 is set to 1 when a WAIT macro instruction referring to this ECB is issued; bit 1 is set to 1 when a POST macro instruction is issued.

<u>Status</u>: Normal status: set to zero. Abnormal status: set to 1.

<u>Statement Number</u>: Number of the statement in which the task was attached.

<u>MCF</u>: Set when the associated task is not in a position to be terminated by a higher level task.

<u>WTF</u>: Set by a higher level task which is about to terminate the task associated with the event variable.

<u>Infinite Wait ECB</u>: Waited on when the task associated with the event variable is about to be terminated by a higher level task.

<u>Wait to Terminate ECB</u>: Waited on by a higher level task when the MCF is on.

```
     0       7 8                            31
     ┌────────┬─────────────────────────────┐
   0 │ Flags  │      Length of PRV VDA      │
     ├────────┴─────────────────────────────┤
   4 │         A(External save area)        │
     ├──────────────────────────────────────┤
   8 │                                      │
     │ Pseudo-register vector (PRV)         │
     │                                      │
     ├──────────────────────────────────────┤
     │         A(Attaching DSA)             │
     ├──────────────────────────────────────┤
     │         A(Attaching PRV VDA)         │
     ├──────────────────────────────────────┤
     │         A(Task variable)             │
     ├──────────────────────────────────────┤
     │         A(Parameter list)            │
     ├──────────────────────────────────────┤
     │ Optional entries:                    │
     │           ON field                   │
     │           Parameter list             │
     │                                      │
     ├──────────────────────────────────────┤
     │ Library workspace (LWS)              │
     └──────────────────────────────────────┘
```

Figure 78. Format of PRV VDA for Multitasking

A PRV VDA for multitasking is identified by a 1 in the first bit of the length field (bit 8 of the PRV VDA). Like its non-multitasking counterpart (Appendix J), it contains the PRV and primary LWS and is chained back to the external save area. It differs in the settings of the flag byte and in the presence of the following additional fields immediately following the PRV:

1st word: Chain back to the DSA of the attaching task.
2nd word: Chain back to the PRV VDA of the attaching task.
3rd word: Address of its own task variable.
4th word: Address of the parameter list for the called procedure; if no parameters are being passed, this word is set to zero.

The following fields are omitted if there are no entries:

ON field: When a subtask is attached, the entries in the ON field of the DSA of the attaching task are copied into this field.
Parameter list: Parameter list for the called procedure.

The settings of the flag byte are as follows:

Major task	X'29'
Subtask	X'2D'
Subtask with entries in ON field	X'2F'

TASK VARIABLE

	0 7	8 15	16 31
0	Flags	A(PRV VDA)	
4		A(TCB)	
8		A(SYMTAB entry)	
C		A(Event variable)	
10	Limit priority		Dispatching priority
14		Chain-forward address	
18		Chain-back address	

Figure 79. Format of the Task Variable

The task variable contains the task control information required by the PL/I Library. To enable subtasks to be detached when the attaching task is terminated, all task variables activated in a task are placed in a chain anchored in the DSA of the attaching task. Only the first two bits of the flag bytes are used:

Bit 1:	0 =	Task variable inactive (task not attached)
	1 =	Task variable active
Bit 2:	0 =	CALL with TASK option
	1 =	CALL without TASK option

IBM System/360 Printed in U.S.A. Y28-6801-4

International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, N.Y. 10601
[USA Only]

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
[International]