File No. S360-29
GY28-2052-0

Program Logic

IBM System/360 Time Sharing System
PL/I Subroutine Library

Program Number 360S-LM-512

This publication describes the internal specifications of PL/I Subroutine Library as a system component of IBM System/360 Time Sharing System. The relationships between the code produced by the TSS/360 PL/I 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.

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 Operating System:

- Principles of Operation, Form A22-6821
- PL/I Language Specification, Form Y33-6003

IBM System/360 Time Sharing System:

- Assembler Language, Form C28-2000
- Concepts and Facilities, Form C28-2003
- TSS/360 PL/I Reference Manual, Form C28-2045
- System Programmer's Guide, Form C28-2008
- PL/I Library: Computational Subroutines, Form C28-2046
- TSS/360 PL/I Programmer's Guide, Form C28-2049
- System Control Blocks, Form Y28-2011
- Assembler User Macro Instructions, Form C28-2004
- TSS/360 PL/I Compiler, Program Logic Manual, Form Y28-2051

An introductory section, 'The PL/I Library' and the first part of Section II contain a general description of the library as a component of IBM System/360 Time Sharing System, and general notes on features of the system and the TSS/360 PL/I Compiler that are used in the library implementation. The remainder of Section II 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, data control block descriptions, and several appendixes. The module summaries, in Section III, 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 fourth section contains detailed descriptions of the format and content of the control blocks used by the PL/I Compiler. The appendixes contain details of the system macro instructions used, library pseudo-registers and macro instructions, and library internal error codes.

First Edition (June 1970)

This edition applies to Version 7, Modification 0, of IBM System/360 Time Sharing System, and to all subsequent releases until otherwise indicated in new editions or Technical Newsletters. Changes are periodically made to the specifications herein; before using this publication in connection with the operation of IBM systems, refer to the latest edition of IBM System/360 Time Sharing System: Addendum, Order No. GC28-2043, for the editions of publications that are applicable and current.

This publication was prepared for production using an IBM computer to update the text and to control the page and line format. Page impressions for photo-offset printing were obtained from an IBM 1403 Printer using a special printing chain.

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

A form is provided at the back of this publication for reader's comments. If the form has been removed, comments may be addressed to IBM Corporation, Time Sharing System/360 Programming Publications, Department 636, Neighborhood Road, Kingston, New York 12401.

FIGURES

FUNCTION

The PL/I library was designed as a set of reentrant object modules, each performing a single function or a group of related functions.

The library modules can be divided into two groups:

1. Those that act as an interface between compiled code and the IBM System/360 Time Sharing System; these modules are mainly concerned with input/output, dynamic program and storage management, and error and interrupt handling.

2. Those that are closed subroutines specifically designed to perform arithmetic computations, data conversions, I/O editing and string generic built-in functions as the major part of their task.

USAGE

The code produced by the PL/I compiler includes many calls to PL/I Library modules, where their specific functions are required. The library modules themselves reside in SYSLIB, and are dynamically loaded when referenced during program execution. The modules explicitly called by object code will remain in virtual storage until the user unloads his object module explicitly. Thus, succeeding executions of the same module will not cause reloading of library modules.

The PL/I library acts as the sole interface between compiled code and the system. The compiled code does not issue SVCs or system macro instructions but instead issues a library call. Although the library module(s) called can issue an SVC instruction, it is more convenient to use system macro instructions. This method means that when the system changes, only the library module is rewritten, with the call to the library from the compiler remaining as before. Similarly, if the SVC calling sequence changes, the system macro is changed accordingly and the library module need only be reassembled.

For further details on macro instructions, see IBM System/360 Time Sharing System: Assembler User Macro Instructions. The system macro instructions used by the library are listed in Appendix A.

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.

Under TSS, PL/I statements that are related to a number of facilities will be accepted for compilation but are not supported and will, during execution of the PL/I program, cause a diagnostic message to be issued. The unsupported facilities, and their associated system actions, are:

Facility	System Action
Checkpoint	Continue
Multitasking	Revert to command mode
Sort/Merge	Revert to command mode
Restart	Continue or Finish condition
Teleprocessing	Finish condition
Regional I/O	Finish condition

SECTION I

METHOD OF OPERATION

GENERAL IMPLEMENTATION FEATURES

NAMING CONVENTIONS

External Names

PL/I Library module names always begin with IHEW; other external names always begin with IHE. The uses and meanings of these names are explained in 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 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.

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.

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 system standard is used. One PL/I Library module, IHEWSAP, does not use either of these standards. The subroutines in this module pass arguments by value as well as by name.

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 unchanged:

   Registers RB through LR
   Program mask

   while the following may be changed:

   Registers R0, RA, and BR
   Floating-point registers
   Condition code

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

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

The standard save area is a 76-byte area in which the contents of all the general registers can be saved. The format is described in Section IV.

The library does not support intermodule 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 IHEWSAP. (See 'Library Workspace'.)

LIBRARY MACRO INSTRUCTIONS

Eight macro instructions are available for use in the library modules; they reside in PLIMAC. To obtain these instructions, PLIMAC should be used with PLINDX when assembling. Five of these, IHEYCVC, IHEEVT, IHELIB, IHEZAP and IHEZZZ, set up symbolic definitions and the other three, IHECVC, IHESDR, and IHEPRV, set branches to external modules, and set the current addresses of the standard save area and the pseudo-register vector (PRV) respectively. The library macros are described in Appendix C.

SHARED LIBRARY

With the exceptions of IHEWCVC, one copy of all the PL/I library modules is shared by all TSS users.

Shared modules cannot contain any type of address constant, so these are collected in the non-shared module IHEWCVC. Thus, IHEWCVC contains a list of V-type address constants for every library entry point which can be called from another library module. The order of these entries is primarily alphabetical, but is irregular where certain library modules select V-cons by their position in a table of V-cons.

To invoke a library module, library macro IHECVC is used to load register BR from the list of V-cons in IHEWCVC. This is followed by the normal BALR instruction.

IHEWCVC also contains L-form and executable macros. The macro expansions contain address-dependent parameter lists or address constants. The L-form macros are used with the corresponding E-form: the executable macros have no L-form and are invoked with a BAL instruction.

Finally, IHEWCVC contains miscellaneous address constants from the library.

The base address of IHEWCVC is set in pseudo-register IHEQCTS by initialization in IHESAP. Offsets in IHEWCVC are known to all modules from the DSECT IHEZCVC, which is part of macro IHELIB.

LINKAGE EDITING

A very significant space reduction (twenty-fold) is obtained by combining the library into two modules, CFBAI and CFBAJ, which may reside on SYSLIB. CFBAI contains 30 control sections, all of which are shareable and smaller than one page. CFBAJ contains the un-shared routine IHEWCVC in a single control section.

The arrangement of the original modules is given in Appendix G.

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 'Program Management' later in this section.

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 formats:

Fixed-point binary
Floating-point
Packed decimal

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.

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 system standard is used. One PL/I Library module, IHEWSAP, does not use either of these standards. The subroutines in this module pass arguments by value as well as by name.

Data Type		Implementation			
Scale	Base	Precision	Internal format	Alignment	Processing
REAL data					
Fixed	Binary	p,q Max p: 31	Fixed-point binary	p>15: Word p≤15: Half-word	Arithmetic operations are performed on p-digit integers: scale factor q is specified in a data element descriptor. (See Figure 47)
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: Double-word	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: Double-word	The data is normalized in storage before and after arithmetic operations.
COMPLEX data					
Fixed	Binary	p,q Max p: 31	Fixed-point binary	p>15: Word p≤15: Half-word	As for real fixed binary. The real and imaginary parts occupy adjacent fullwords or halfwords, 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 the real part first.
Float	Binary	p Max p: 53	Hexadecimal floating-point	p≤21: Word p>21: Double-word	As for real floating-point 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: Double-word	As for real floating-point 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 Representation

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 Section IV.)			

Figure 3. String Data Representation

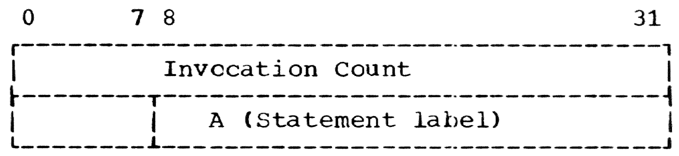


Figure 4. Statement-Label Data Representation

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). Section IV gives the format of LWS, which is allocated by the library program management module IHEWSAP.

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 Section IV; 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 this section and in Section IV.

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

This is a 4096 byte table, consisting of 4 or 8 byte entries called pseudo-registers (PRs). These PRs effectively operate as implicit arguments giving information about, for example, current program status. All references to specific PRs within the PRV are made by the addition of a fixed displacement to the PRV base address contained in register PR.

All PRs used in the PL/I Library are defined as a standard set of 29 in the library macro IHELIB; this macro is coded at the beginning of every library module. The PRs used by the PL/I Library are shown in Appendix B.

<u>Library Workspace (LWS)</u>

Various library modules require working storage:

1. For internal functions.

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

Library modules which use library workspace (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 Section IV.) 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 IHEWSAP controls the allocation of LWS and the setting of the library pseudo-registers. 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 IHEWSAP.

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

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

<u>IHESDR</u>: Used when a module requires a standard save area for a module it is calling.

<u>Library Communication Area (LCA)</u>

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 Section IV. 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 PL/I program. 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.

<u>Execution-Time Dump</u>

A PL/I user may obtain a dump at any point in his program by calling IHEWDUM; the entry point used determines whether the PL/I program will continue or terminate after the dump.

IHEDUMC:	Dump and continue
IHEDUMT:	Dump and terminate

If the program is being run nonconversationally, all pages containing save areas or file blocks will be written on SYSOUT, with DSNAME=PLIDUMP. 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, IHEWZZC and IHEWZZF, 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 E.

If the program is being run conversationally, there will be no automatic dump; an index of the current areas will be printed and a PAUSE issued, to invoke the command mode. The user can then display any of the areas defined in the index, and may return to the PL/I program by the GO command.

INPUT/OUTPUT

FILES AND DATA SETS

Within this publication, the term 'data set' refers to a collection of records that exist on external storage. A file is known as such only within a program.

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 DDEF command). 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 DDEF command 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 accomodate 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 PL/I program. 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 exception 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 Section IV, 'Input/Output Control Blocks'.

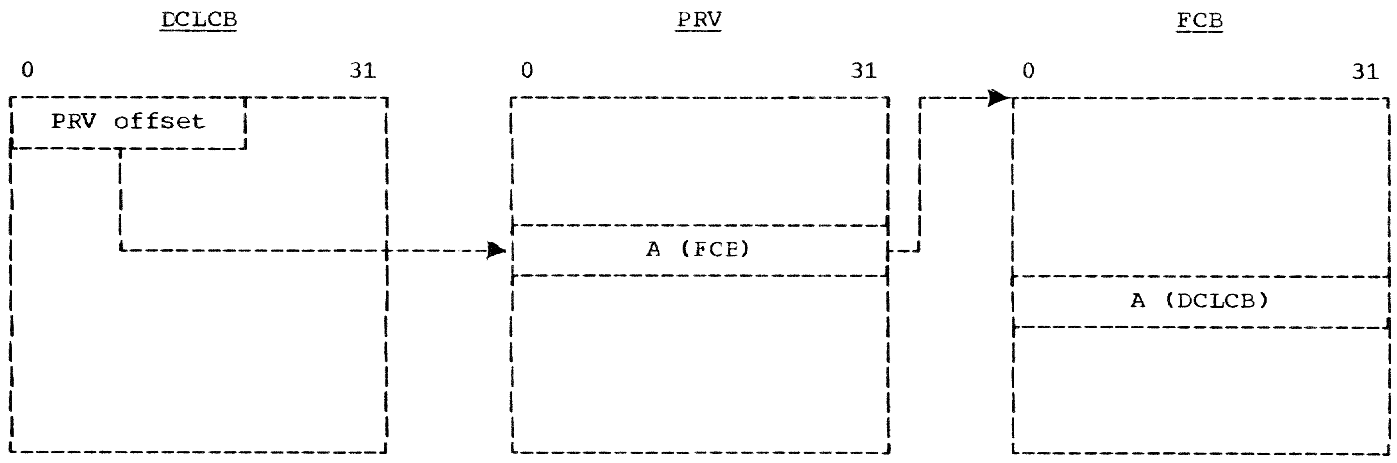


Figure 5. File Addressing Scheme

File Control Block (FCB)

This control block is generated during program execution when a file is opened. The FCB storage is required in order to accommodate reentrant usage of a given PL/I module, for the FCB is not read-only. The FCB contains fields for both the PL/I Library and for TSS/360 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 Section IV; details of the various DCB constructions are available in the following IBM publications:

IBM System/360 Time Sharing System: System Control Blocks

IBM System/360 Time Sharing System: Assembler User Macro Instructions

An FCB is generated for each file opened within a program; an FCB cannot exist for an unopened 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 the PL/I program. 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 program termination (whether normal or abnormal). When a PL/I program terminates, the object-program housekeeping routines determine which files are currently open for this PL/I program. This is performed by the relevant housekeeping module calling IHEWOCL (close), which scans the chain and calls IHECLTB to close all open PL/I program files. If the cell IHEQFOP is zero, then no files are, at present, open for the PL/I program. The IHEQFOP chain is shown in Figure 6.

Program Execution

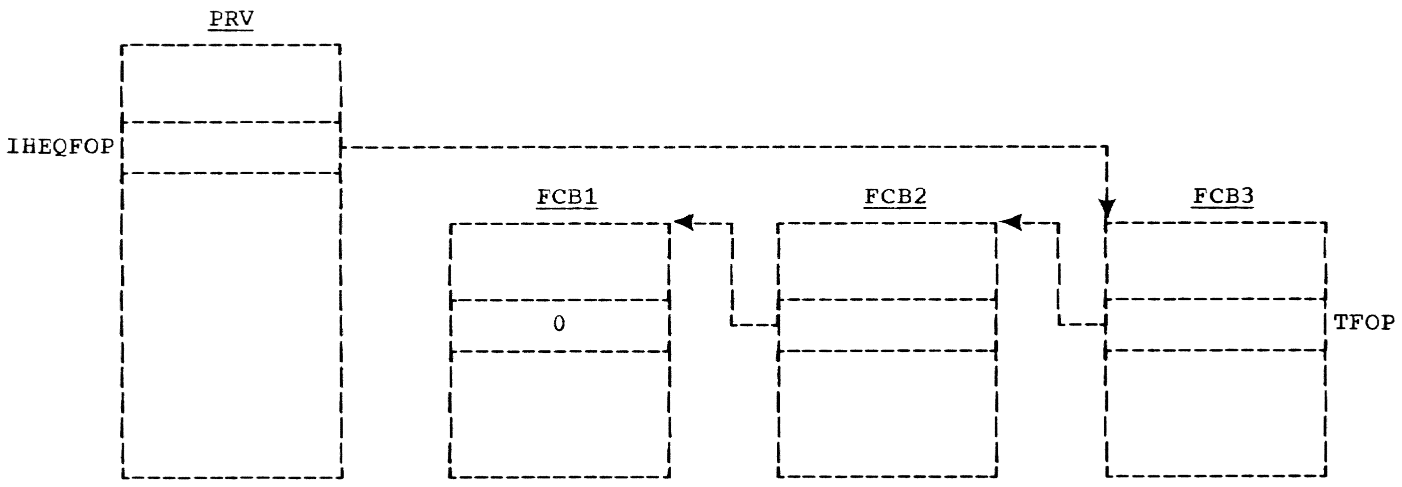
When program execution is initiated, the PRV (including all file registers) is initialized to zero. When a file is opened (prepared for I/O operations), its associated file register is set to address an FCB; similarly, when a file is closed explicitly, its file register is again set to zero.

If the file is not opened, 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.

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. The address of the DCLCB can be used 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.



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

Figure 6. Format of the IHEQFOP Chain

Error code	Meaning
81	Conflict between DECLARE and OPEN attributes
82	File access method not supported
83	No block size
84	No DDEF
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 DDEF 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 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

The modules involved in OPEN processing are IHEWOCL, IHEWOPN, IHEWOPO, IHEWOPP, and IHEWOPQ. These modules are link edited together, and transfer control between one another with either the CALL macro, or a direct branch. The module IHEWOCL passes a list of all necessary address constants and pseudo-register offsets; this list is contained in the library module IHEWSAP.

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

All errors are communicated back to IHEWOCL by means of the file registers; IHEWOCL then invokes the error handling subroutine. The error conditions are signaled in the high-order byte of the file register; IHEWOCL, 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 IHEWOPN 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 Section IV for details of the OCB.)

The Open Process

Open Process, Phase I: IHEWOPN: This performs file attribute checking and defaulting functions. If an attempt is made to open a REGIONAL file, an error message will be issued to SYSOUT; the file will be undefined and the error condition raised. If, in phase I, all files specified in the OPEN statement have detected errors, a return to IHEWOCL is made immediately. Otherwise phases II, III and IV are invoked and a return is made to IHEWOCL from IHEWOPQ.

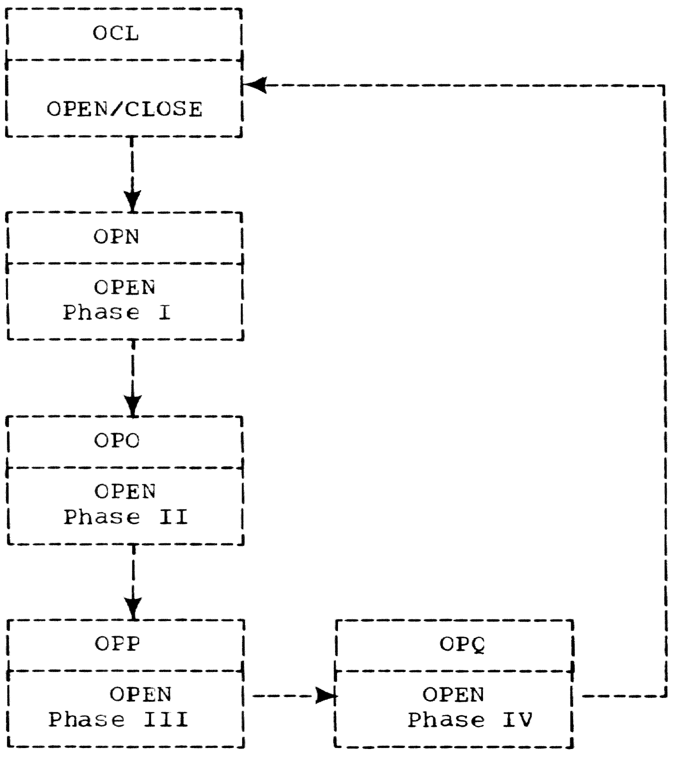


Figure 8. Flow Through the OPEN Modules

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

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

Open Process, Phase IV: IHEWOPQ: This calls 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.

Any files which devolve to the TSS SYSOUT or SYSIN will not need to be opened; however, the checking which is done in these modules is useful, so only the actual OPEN process (in IHEWOPP) will be branched around.

If, during Phase II or Phase III, a file is determined to be a RECORD I/O file, and no JFCB exists for it, a diagnostic will be issued, and return to command mode will be effected by raising the 'FINISH' condition.

The Close Process

This process consists of removing files from the IHEQFOP chain and freeing dynamically acquired storage (file control blocks, buffers, exclusive control blocks, and I/O control blocks).

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

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, IHEWOCL scans the IHEQFOP chain to find the file. For an implicit close, all events associated with I/O event variables in the IHEQEVT chain are purged, and the associated IOCBs, if any, are freed.

Module IHEWCLT performs 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 I/O event variables associated with the file are set complete, abnormal, and inactive, and the I/O operations are purged.

Any files which devolve to the TSS SYSOUT or SYSIN will not need to be closed; however, the checking which is done is useful, so only the actual CLOSE macro (in IHEWCLT) will be branched around.

IMPLICIT OPENING

If a file is not open and an I/O operation is initiated, then one of the compiler-interface modules (IHEWIOA, IHEWIOB, or IHEWION) calls IHEWOCL at implicit-open entry point IHEOCLC, passing any implied parameters, and the open process begins.

If the OPEN modules return control to IHEWOCL and the file is still unopened, the UNDEFINEDFILE condition is raised.

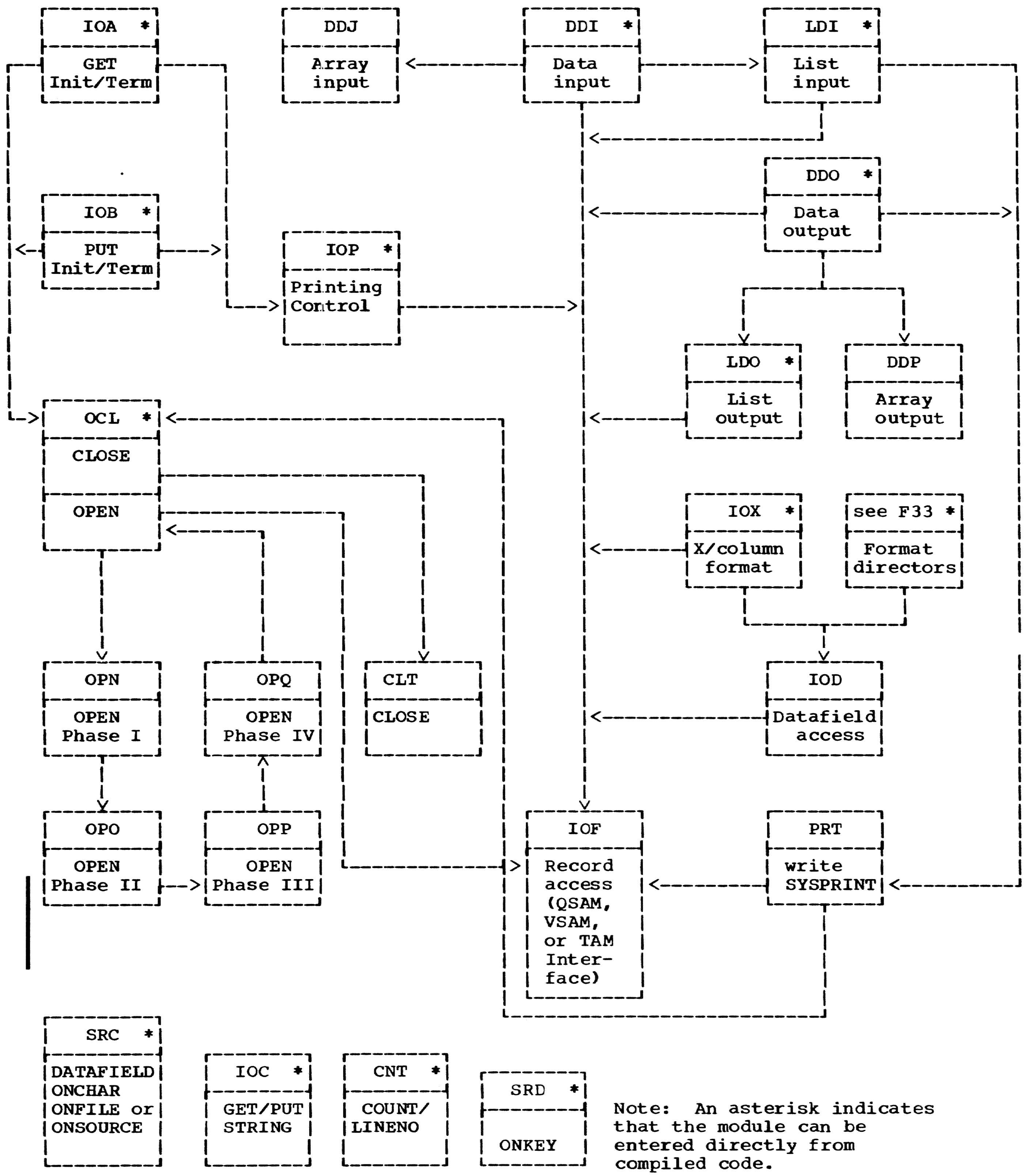
STREAM-ORIENTED I/O (SEE FIGURE 9)

The stream-oriented I/O facilities of PL/I provide for the transmission of data items to or from external storage, without considerations of logical and physical record lengths affecting the user program. These facilities block and deblock user data items in a manner that is transparent to the user, so that they can be read and written by the system's data management routines. When a record area becomes filled (on output) or empty (on input), the user data is continued on another record; the user will never be aware of this break.

Support for record access is provided by the data management VAM and QSAM routines, and SYSIN and GATWR macro instructions. The VSAM and QSAM GET and PUT macro instructions used are all locate mode, to conserve space and time; SYSIN and GATWR macro instructions are used when the file is SYSIN or SYSOUT, respectively.

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 10.



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

Figure 9. Modular Linkage Through Stream-Oriented I/O

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

Figure 10. 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 item in the data stream are flagged in the fifth byte of IHEQCFL:

TRANSMIT to be raised on item:	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., IHEWIOA and IHEWIOB), 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. 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

Within the PL/I language, the keywords SYSIN and SYSPRINT indicate the standard input and output files. At execution time, SYSIN is interpreted as the TSS SYSIN, and SYSPRINT as the TSS SYSOUT.

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 the error messages, and listing of data fields for the COPY and CHECK options require the presence of this file.

Because the library and the source program both use the SYSPRINT file, it is necessary that they both refer to the same DCLCB. Both the compiled DCLCB and the library-supplied DCLCB for SYSPRINT (within the module IHEWPRT) 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.

If the current file is SYSIN, a SYSIN macro is issued to obtain stream input; this will send a colon to SYSIN, to indicate to the user that input is expected. The input data will be placed in the buffer provided for this file.

If the current file is SYSPRINT and the task is conversational, the data specified by the user in his PUT statement will be immediately written out to his terminal by a GATWR macro instruction. This data will be followed by a return suppression character, so that the user's terminal will continue typing the next record on the same line, unless the SKIP option is specified in the next PUT statement.

If the current file is SYSPRINT and the task is nonconversational, the data specified by the user in his PUT statement will be placed in the output buffer, but will not be written out until that buffer is full, or until the SKIP option is specified in a PUT statement.

The SYSIN and GATWR macros required for SYSIN and SYSPRINT I/O are contained in library modules IHEWIOF and IHEWIOB.

Get/Put Object Program Structure

The code compiled for stream-oriented I/O GET and PUT source statements has the general structure illustrated in Figure 11. 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.

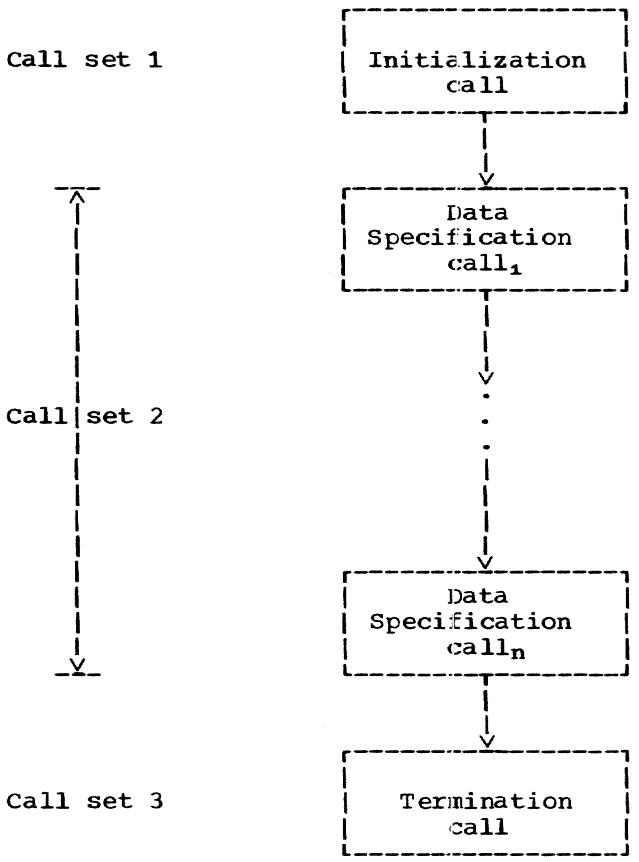


Figure 11. Object Program Structure of GET/PUT

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' earlier in this section.) The PL/I standard (using registers) is employed wherever possible; where it is not, the 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' later in this section.)

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 12. The first column, 'Primary code', consists of calls to units in the second column, 'Secondary code'; that is, 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.

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 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.

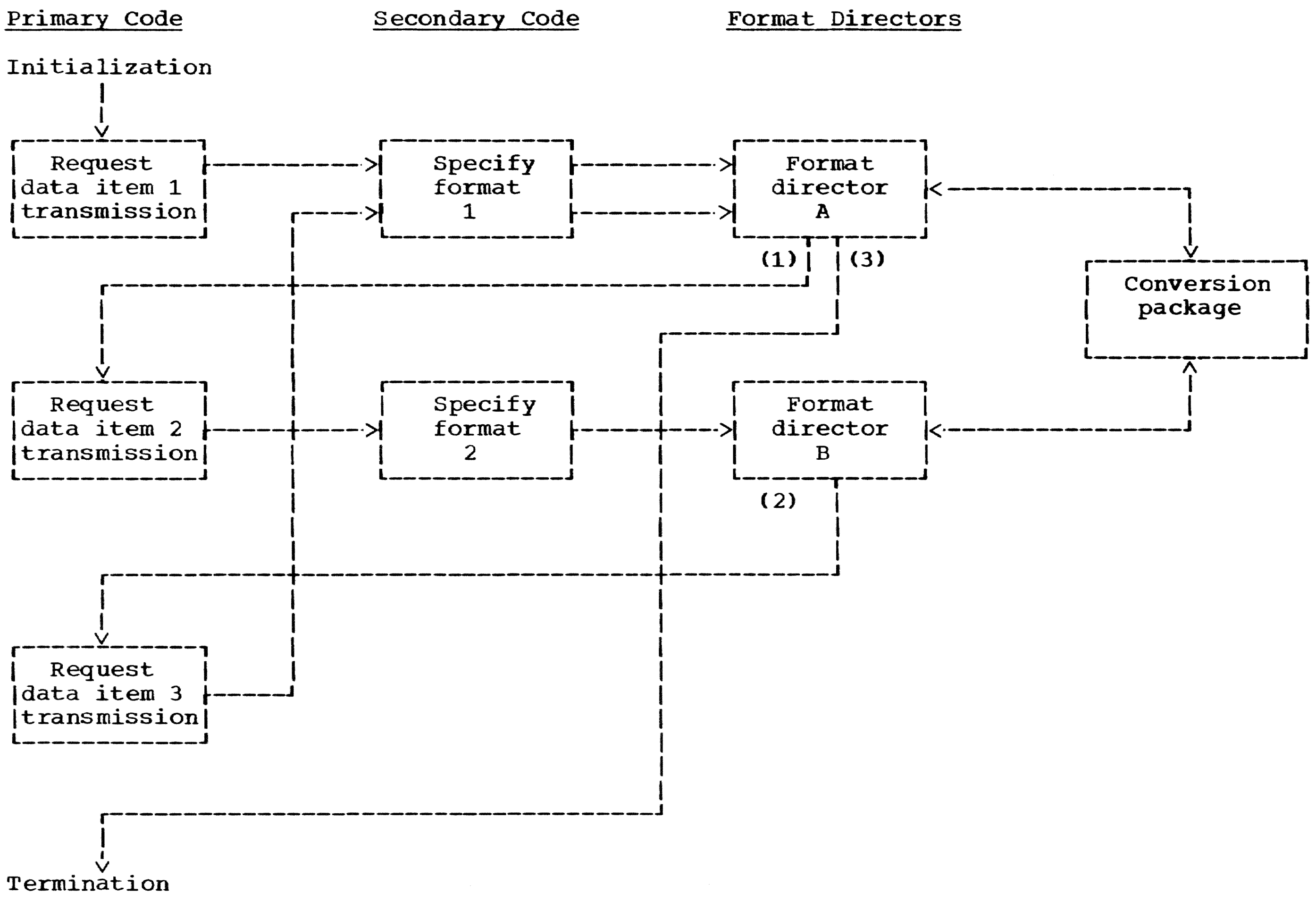


Figure 12. Executable Format Scheme

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, SYSOUT. This is performed by calling the module IHEWPRT. Each data field occupies the initial portion of a line

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 (IHEWIOC) 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 IHEWIOP. All printing options (and format items) are supported by use of the American National Standard FORTRAN control characters:

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

For conversational tasks, PAGE will be treated as a triple space; a SKIP of more than 3 in an output file will be treated as a triple space; and a LINE request which implies more than 3 spaces will be treated as a triple space.

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.

   If the task is conversational, and SYSIN is the input file, then the SKIP option is 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 (for conversational tasks, a maximum of 3 lines will be inserted). 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.

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, independent of record boundaries).

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 13, 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'.

ORGANIZATION	ACCESS	MODE	BUFFERING	RECORD FORMAT	ACCESS METHOD
CONSECUTIVE	SEQUENTIAL	INPUT OUTPUT UPDATE	BUFFERED	ALL	QSAM/VSAM
			UNBUFFERED	F, U, V	BSAM
INDEXED	SEQUENTIAL	INPUT OUTPUT UPDATE	BUFFERED or UNBUFFERED	F, V	VISAM (GET/PUT)
	DIRECT	INPUT UPDATE	UNBUFFERED	F, V	VISAM (READ/WRITE)

Figure 13. Data Management Access Methods for Record-Oriented I/O

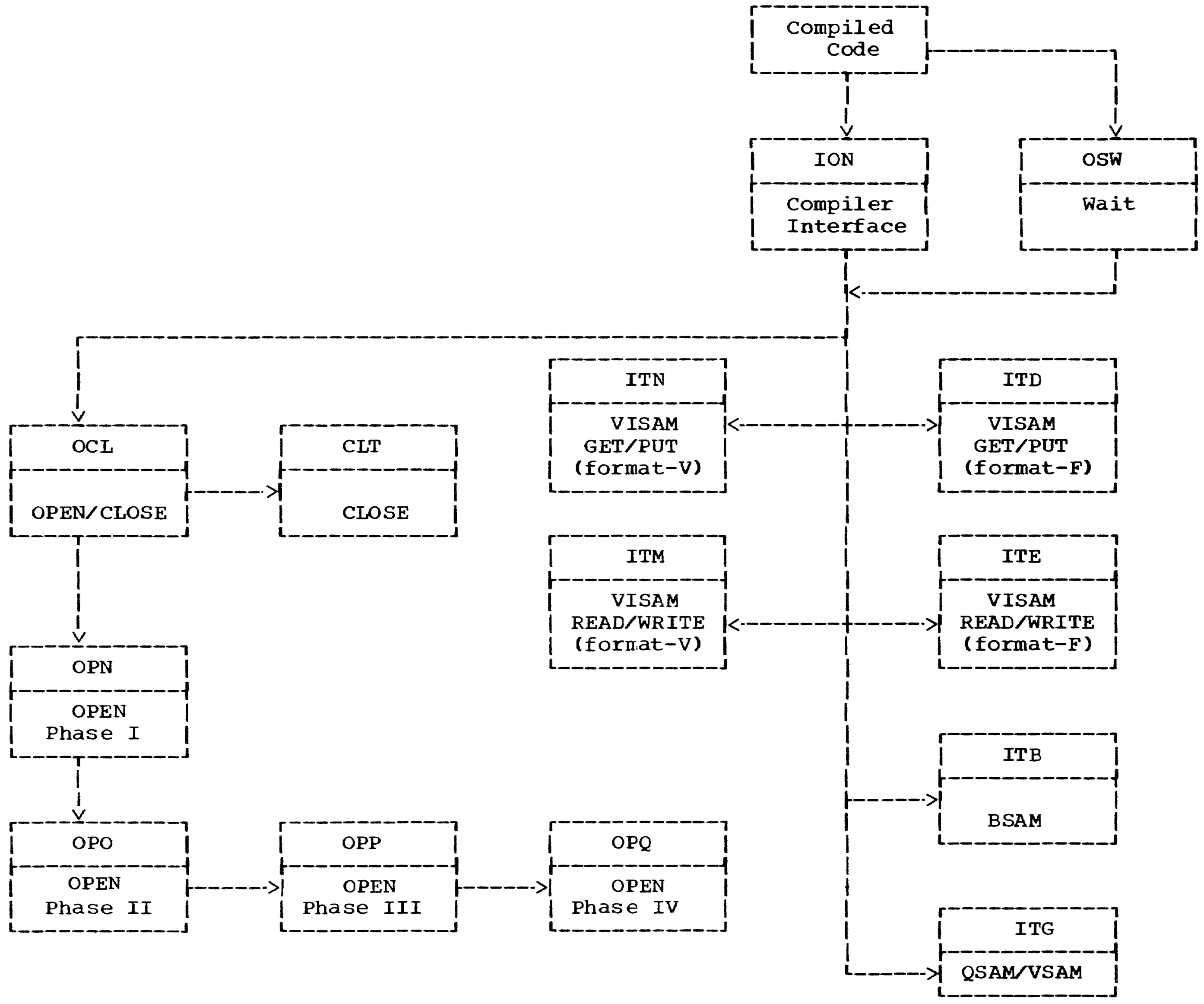


Figure 14. Linkage of Access Modules in Record-Oriented I/O

General Logic and Flow

The overall flow of record-oriented I/O modules is illustrated in Figure 14. Module IHEWION is a general interface module which is invoked by a compiled call for any record-oriented I/O statement. This module interprets the requested I/O operation, verifies its applicability to the specified file (and, possibly, implicilty opens it), and then invokes an access method interface module (characterized by the module names IHEWIT*) to have the operation performed.

The verification of a statement is performed by IHEWION 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 IHEWOPQ 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 and BSAM is the access method) after initiation of transmission, control returns to the general interface module IHEWION, and thence to the compiled program. Errors may be detected within IHEWION 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 IHEWION 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 Section IV) is associated with a record-oriented I/O operation, invokes the relevant I/O transmitter (IHEWIT*), 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, if the access method was BSAM, and performs any checking required before returning control to the WAIT module.

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 IHEWION from the FCB associated with the file; the field TACM in the FCB is set during the open process to point to the appropriate 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 Section IV.)

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), 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 (VISAM):
The IOCBs are created by IHEWITE. 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 Section IV.

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.

IOCB -- SEQUENTIAL Usage: Manipulation of IOCBs for SEQUENTIAL usage is required only for BSAM, which is employed for CONSECUTIVE UNBUFFERED files.

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 TIAB field in the FCB points to the last IOCB to be used.

The chain of IOCBs is required because all I/O operations must be checked in the order in which they were issued. This chain 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 DDEF subparameter NCP. The value of this subparameter should not be greater than 1 unless the EVENT option is employed; if NCP is unspecified a default of 1 is used. If NCP = 1, there is then one IOCB and one channel program.

The size of each IOCB varies, depending upon the organization and the record format of the data set. Section IV specifies the size requirements.

IOCB -- DIRECT Usage: Manipulation of IOCBs for DIRECT usage is required for VISAM. One IOCB is allocated to a DIRECT file when it is opened; subsequent selection of an IOCB is performed by IHEWITE. Unlike SEQUENTIAL access, the order of I/O operation is not normally considered.

The chain of IOCBs for a given file is anchored in the TLAB field in the FCB. The chain is released when the file is closed.

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:

1. QSAM or VSAM
2. BSAM

The choice between them is governed by the file attributes BUFFERED and UNBUFFERED:

BUFFERED:	QSAM or VSAM
UNBUFFERED:	BSAM (F,V,U) (No automatic blocking or deblocking)

The choice between VSAM and QSAM for BUFFERED files is determined by the data set organization, as specified in the DDEF.

QSAM/VSAM (IHEWITG): 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 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 IHEWITG 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.

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 system uses the ABEND macro to end the PL/I program.

BSAM (IHEWITB): 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 System Control Blocks PLM for details of the DECB. IHEWITB 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 method employed for this organization is VISAM.

All usage of INDEXED data sets requires the presence of buffers, even though the file is UNBUFFERED or DIRECT.

VISAM SEQUENTIAL Access (IHEWITD and IHEWITN): SEQUENTIAL creation and access of INDEXED data sets is performed by IHEWITD for format-F records, and by IHEWITN for format-V records. 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, VISAM 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 (that is, 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 (that is, on termination of a PL/I program), 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 on the task's SYSOUT.

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 WRITE (type KS) macro instruction is issued. If the key has changed, the WRITE macro is not issued and the KEY (specification) condition is raised.

To support the DELETE statement without the KEY option, the DELREC macro instruction is used.

If the file has the KEYED attribute, and the mode is INPUT or UPDATE, the VISAM 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 VISAM 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 VISAM 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 DCB-SYNAD field before issuing the CLOSE macro. The system uses the ABEND macro to terminate the PL/I program.

<u>VISAM DIRECT Access (IHEWITE and IHEWITM):</u> Direct access of INDEXED files is performed by module IHEWITE for format-F records, and by IHEWITM for format-V records.

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

Since a VISAM may destroy the contents of the key and record fields when adding new records to the data set, the key value is moved into the buffer before VISAM is invoked. Truncation or padding of the character-string key to conform to the KEYLEN specification is performed during the move.

A DELETE statement is implemented by the DELREC macro instruction.

THE WAIT STATEMENT

Under TSS/360, the WAIT statement is supported for awaiting the completion of I/O events. When the WAIT statement is executed, compiled code calls the library module IHEWOSW, passing the addresses of the event variables associated with the statement. If the required number of events have been completed, return is made to the user; if not, CHECK macros are issued to await the completion of the necessary I/O operations.

Since QSAM, VSAM, and VISAM I/O operations are checked in line (i.e., as they are issued), and are therefore complete before the user receives control after having issued them, a WAIT statement issued for an I/O operation performed by any of these access methods will always find the event completed; only in the case of a BSAM operation may the WAIT statement find the event incomplete, and therefore be required to await completion.

PL/I OBJECT PROGRAM MANAGEMENT

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 IHEWSAP, whose entry points are listed in Figure 15; full details are given in Section III.

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 on entry to the program. Its functions are:

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 15. IHEWSAP Entry Points

1. Allocation of storage for the PRV. (See 'Communications Conventions' in this section.)

2. Initial allocation of LWS.

3. Passing the address of the library error-handling subroutine (IHEWERR), which assumes control when a program interruption occurs, to the system.

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 TSS/360, virtual 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 provide an interface between compiled code and the control program.

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

Time Sharing System Facilities

The following facilities are provided by TSS/360. (See IBM System/360 Time Sharing System: Assembler User Macro Instructions.)

<u>SPEC macro instruction</u>: States the address of an interrupt handling routine with entry point IHEERRA, and indicates that it is to handle program interrupt types 1 to 13, and 15.

<u>SIR macro instruction</u>: Directs control of program interrupt handling to a routine specified in a SPEC macro instruction.

<u>DIR macro instruction</u>: Deletes control references to the routine previously defined to the system by the SIR macro instruction.

<u>GATWR macro instruction</u>: Used to send certain error messages to the user's SYSOUT.

<u>GETMAIN macro instruction</u>: Requests the allocation of a contiguous block of virtual storage to the user's task.

<u>FREEMAIN macro instruction</u>: Releases a specified virtual storage area from a user's task.

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 Section IV.

<u>Dynamic Storage Area (DSA)</u>

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 Section IV.

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 <u>display</u>. 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 <u>invocation count</u>, 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.

<u>Variable Data Area (VDA)</u>

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 Section IV.

<u>Library Workspace (LWS)</u>

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. Section IV 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 system requests (i.e., GETMAINs) necessary to obtain automatic storage, a page of storage is obtained every time a GETMAIN is issued. Areas are allocated by the library from this page (or block) as required until a request is made that is too big to be satisfied from the remaining storage in the block, then another page is obtained. 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, sixteen pages of storage are allocated for the PRV VDA. 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, another sixteen pages are obtained by means of a GETMAIN macro. These blocks are chained to the existing blocks by the free-storage chain. (See Figure 16.)

In any block that contains unallocated storage (that is, contains free storage), 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 storage length in previous block)

<u>4th word</u>: A (Free storage length of following block)

The first and last blocks require a slightly different usage:

<u>First block</u>: Uses the free-storage pseudo-register IHEQSFC in the chaining forward and back:

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

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

<u>Last block</u>: 4th word contains 0

When a request for storage is received, a search of the free-storage lengths, starting from the first, is made. If a free-storage length equal to or greater than the length requested is found, the request is satisfied from that block. The free-storage 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-storage field in the corresponding block is updated. If the page becomes available, it is freed by issuing a FREEMAIN macro, and the free-storage 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

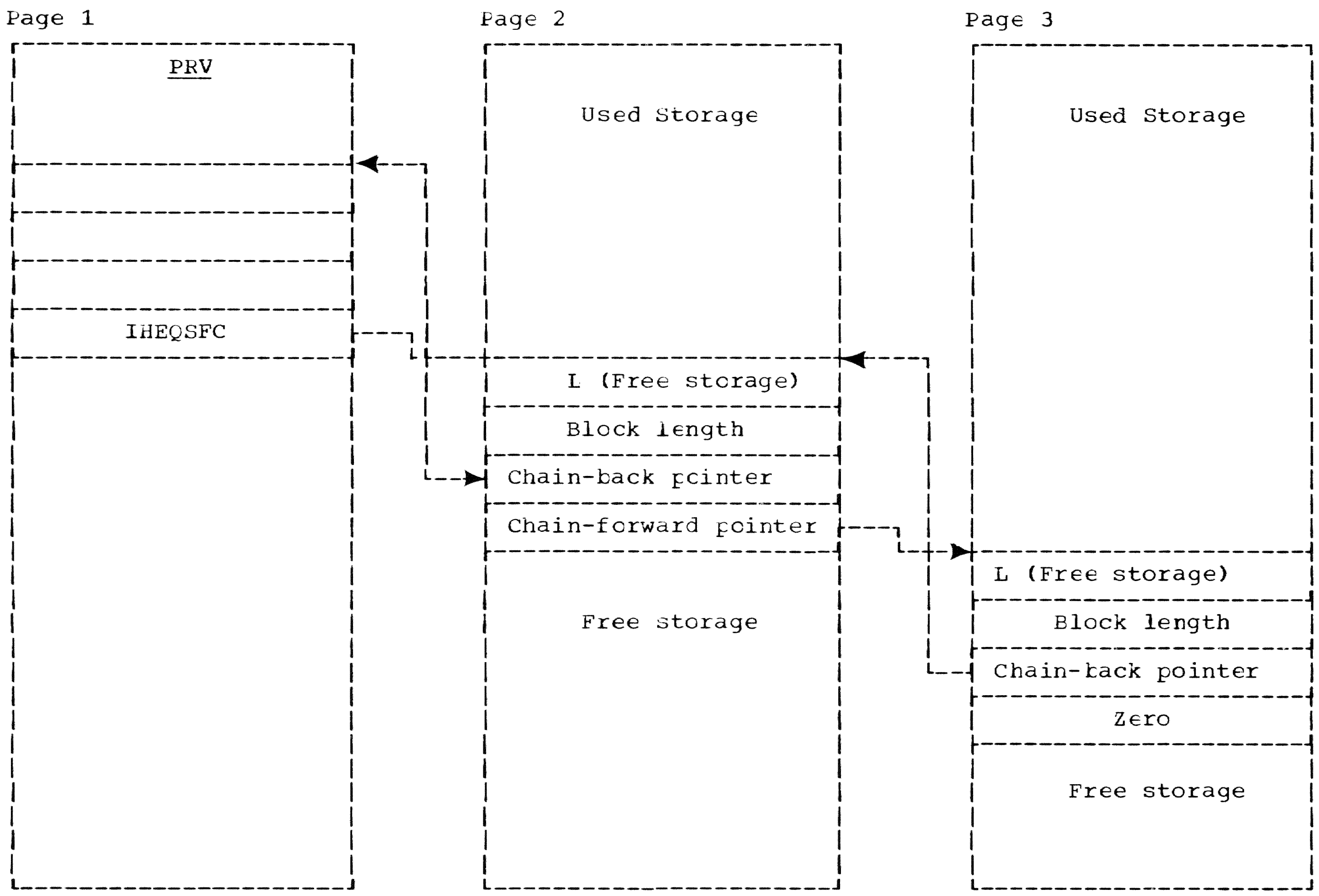


Figure 16. Structure of the Free-Storage Chain for Automatic Variables

when the current allocation is freed (Figure 17). The length of each allocation is recorded in the fullword field following the chain-back address. The length of the data is 12 bytes less than the length of the allocation. 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,

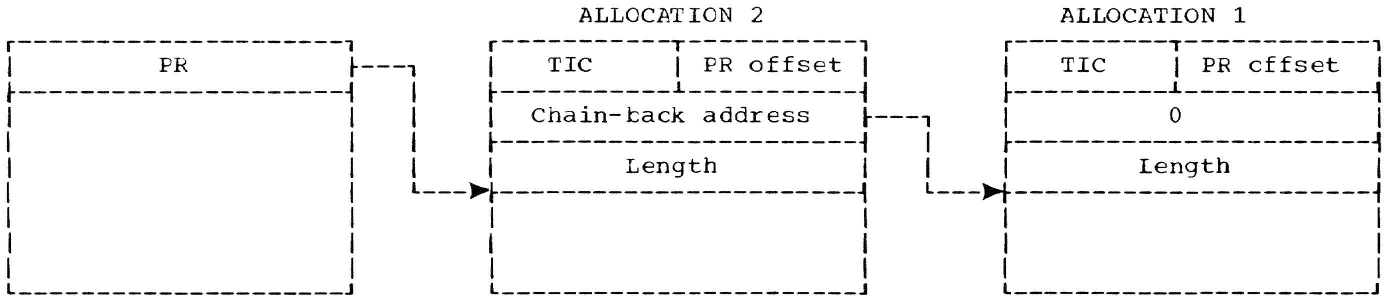


Figure 17. Storage Allocation for a Controlled Variable

for control purposes, and this new length is rounded up to a page. 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 IHEWLSP, 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 PL/I program is freed at the end of the PL/I program.

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 virtual 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 18. 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 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

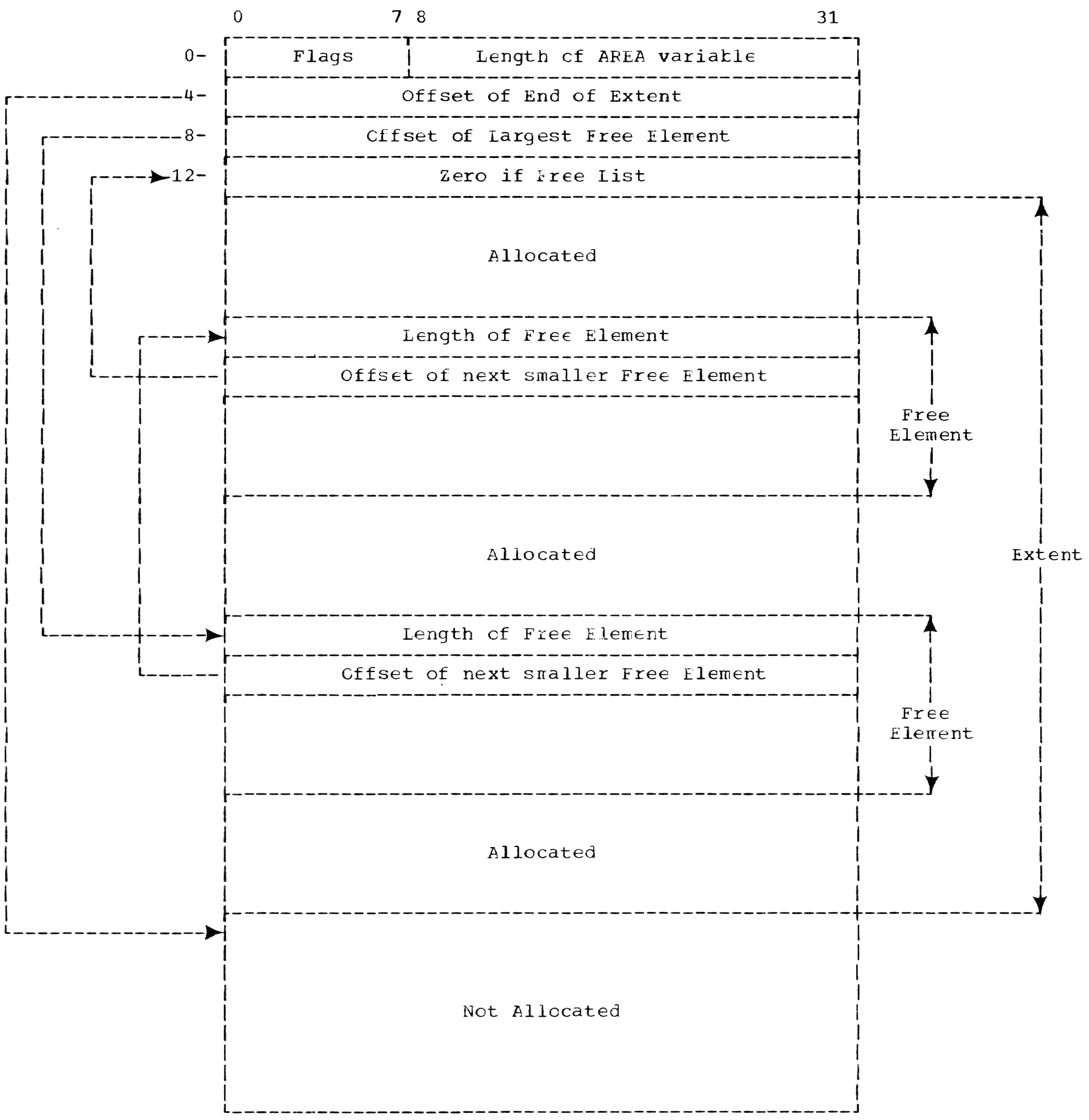


Figure 18. Format of Area Variable

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, IHEWERR 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 and on whether the PL/I program is called from an assembler-language routine. The initialization routine first obtains one page of virtual storage for the PRV VDA.

The initialization routine zeros the PRV, sets up the LWS pseudo-registers, and issues SPEC and SIR macro instructions naming IHEWERR. In addition, IHESAPA and IHESAPC enable a parameter on the statement invoking the PL/I program to be passed to the program. 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 (IHEWERR). 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 freed. 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 execution-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 (IHEWERR) calls IHESAFQ, which closes all the open files in the manner described above; IHESAFQ returns to the supervisor with a return code (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:

1. 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.

Block Housekeeping

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.

DSA Chain: 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 19. If the begin block required a VDA this would be added to the end of the chain. Figure 20 shows an example in which the begin block required two VDAs. If the program now executes:

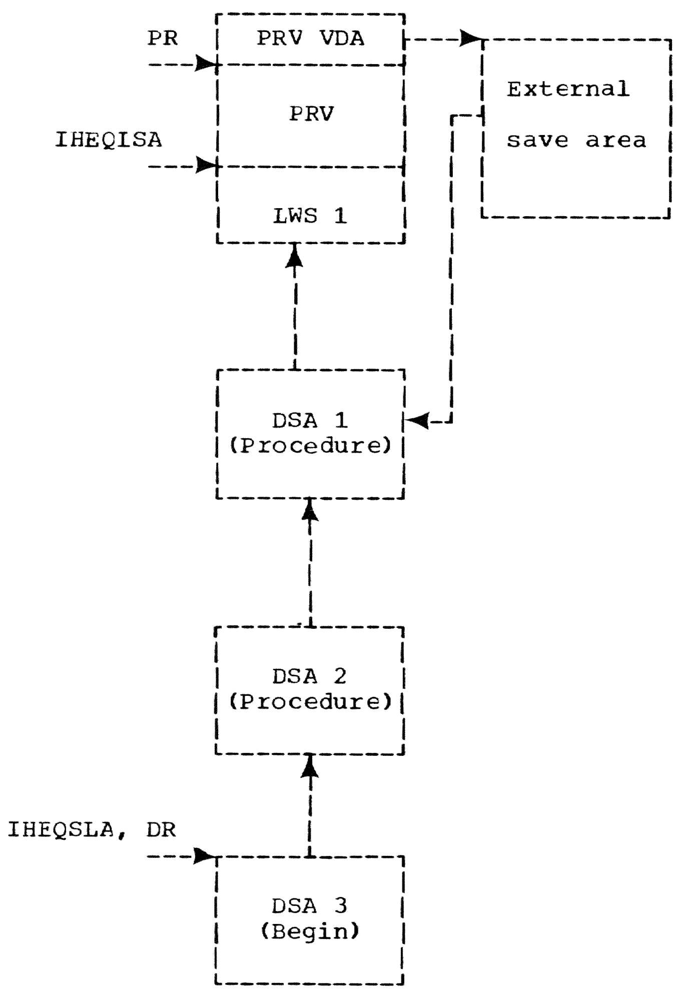


Figure 19. Example of DSA Chain

1. An END statement: 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. A RETURN statement: All areas up to and including the immediately encompassing procedure DSA are released, leaving only items 1 and 2.

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 20, 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:

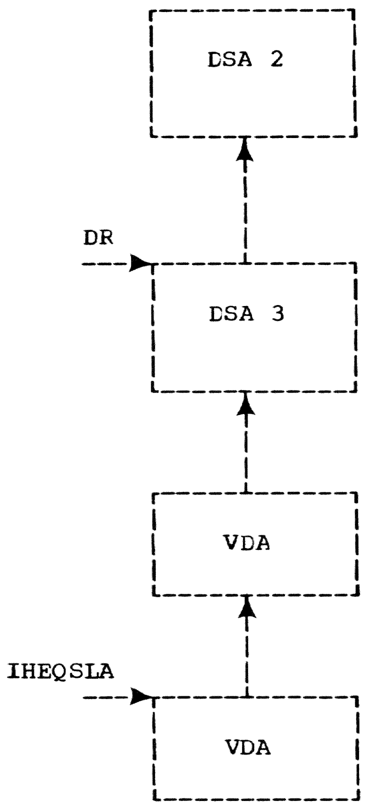


Figure 20. Continuation of the DSA Chain

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 Section IV.)

<u>Save-Area Chain</u>: 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 offshoot of the DSA chain. (See Figure 21.) 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.

<u>Treatment of Interruptions</u>: When a program interruption occurs in a subroutine (library or compiled code), the library error-handling subroutine (IHEWERR) is entered and the address of the save area of that subroutine is set in register DR. (See Figure 22.)

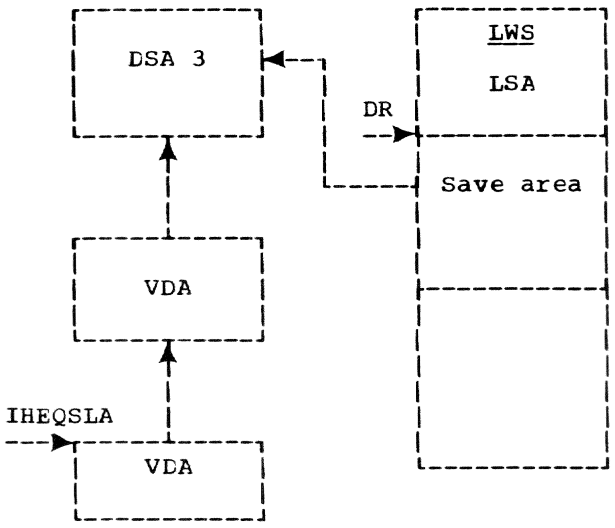


Figure 21. Construction of the Save-area Chain

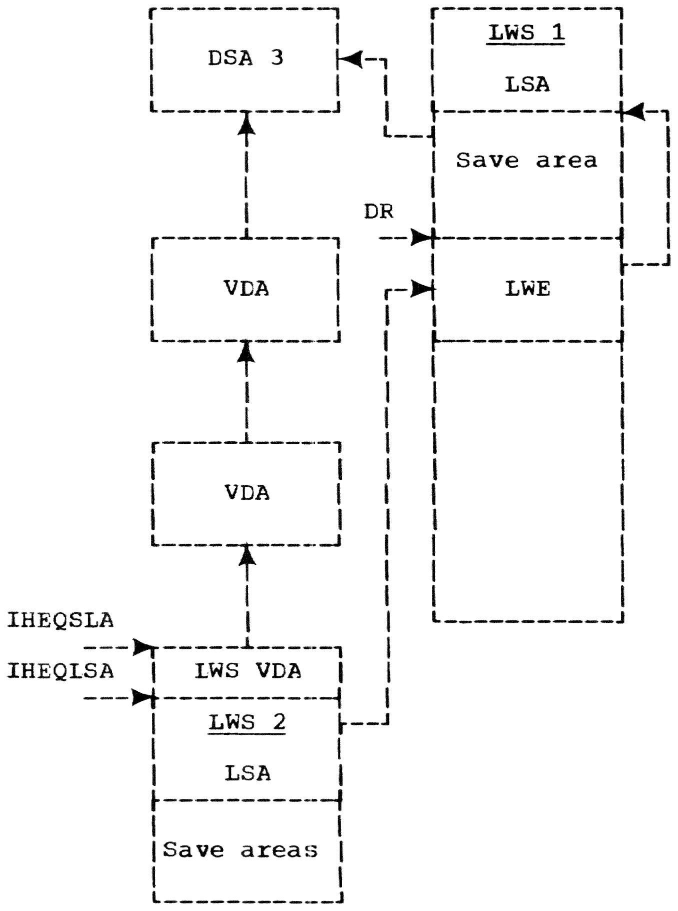


Figure 22. Structure of the DSA Chain When the Error-Handling Subroutine is Entered After a New LWS Has Been Obtained

IHEWERR 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, IHEWERR 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 23.)

If there is no on-unit corresponding to the interrupt condition, standard system action is taken. (See 'Error and Interrupt Handling'.)

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.

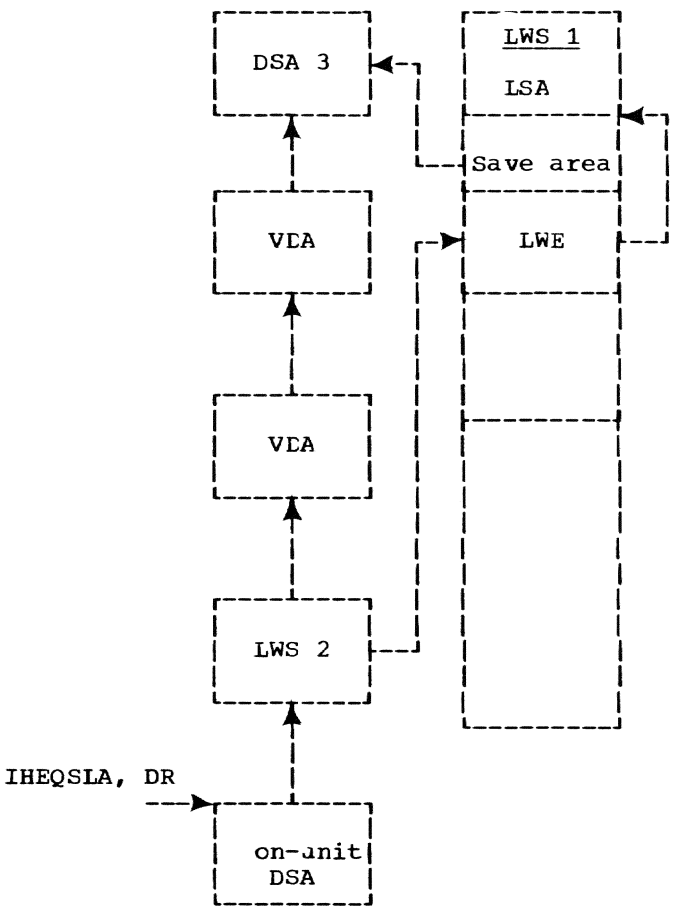


Figure 23. Structure of the DSA chain When the On-Unit DSA is Attached

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 21.

Execution-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, a 512-byte storage area is reserved at the end of primary LWS during initialization and pseudo-register IHEQSLA is set to the address of that save area. 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 Time Sharing System: PL/I Compiler, Program Logic Manual for a discussion of DSAs in STATIC storage.)

ERROR AND INTERRUPT HANDLING

The PL/I Library handles two types of conditions which cause interruption in the main flow of a program during execution:

1. ON conditions, for which it is possible to specify an on-unit.

2. Execution error conditions, for which it is not possible to specify an on-unit.

All the conditions are listed in IBM System/360 Time Sharing System: PL/I Reference Manual.

When any of the conditions occur, control is passed to the library error handling module IHEWERR. Figure 24 describes the flow through this module.

Except for program interruptions, which are discussed separately, all the conditions are raised by compiled code within the PL/I program. When the condition is recognized (e.g., an end-of-file return code from an attempt to read), the PL/I program enters IHEWERR at the entry point appropriate to the condition:

IHEERRB:	ON conditions
IHEERRC:	Non-ON conditions
IHEERRD:	CHECK and CONDITION

If the condition is an ON condition, an internal error code is placed in the pseudo-register IHEQERR before IHEERRB is entered. The codes associated with each ON condition are given in Appendix D.

STANDARD 'SYSTEM' ACTION

If an ON condition is raised and there is a matching ON field for the condition, IHEWERR passes control to the on-unit via IHESARA. Standard 'system' action is taken if: (1) an ON condition is raised and there is no matching ON field for the condition; (2) the system action flag is set in the matching ON field; or (3) an execution error or interrupt condition arises for which no specific ON condition is defined in PL/I (e.g., logarithm of a negative number).

Standard 'system' action consists of printing an error message and, possibly, raising the ERROR condition. The error messages for the conditions are in PL/I Programmer's Guide. After the error message is printed, either processing continues or the ERROR condition is raised, depending on the severity of the error. Raising the ERROR condition usually leads to the FINISH condition being raised. Unless there is a GO TO statement in the ERROR or FINISH unit, the program will return to command mode.

When SNAP or PL/I 'system' action messages must be printed, IHEWERR calls the library module IHEWESM, which contains the code needed to print SNAP and PL/I 'system' action messages. These library error handling modules contain error messages:

IHEWERD:	Data processing error messages.
IHEWERE:	Error messages other than those in the other error message modules.
IHEWERI:	Input/output error messages for non-ON conditions.
IHEWERO:	Error messages for non-I/O ON conditions.
IHEWERP:	Error messages for I/O ON conditions.

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.

PROGRAM INTERRUPTIONS

There are fifteen possible program interruptions in TSS; the PL/I Library handles all but significance. Seven of the interruptions are related to computional ON conditions in PL/I; the remaining seven are treated as errors of a non-ON type (see Figure 25).

The PL/I Library gains control of program interruptions by a SPEC macro instruction, assembled as part of the GET PRV subroutine of the IHEWSAP library module. The SPEC macro specifies which program interruptions will be processed by the PL/I Library (all except significance) and specifies IHEERRA as the entry point of the error handling routine. As a result of the SPEC macro, an Interrupt Control Block (ICB) is created, which contains the program interrupt mask and the address of IHEERRA.

Then IHEWSAP issues a SIR macro, which specifies that IHEERRA is the entry point for the current interrupt handling routine (by giving the name of the ICB associated with it) and sets the priority for the interrupt routine.

Upon entry to the interrupt handling routine at IHEERRA, register 0 is pointing to the fifth word in a 64 word entry used by the system (see Figure 26). The 25thand 26th words of the entry contain the old VPSW, which contains the interrupt code and the address in the PL/I program where control should return when the interrupt has been handled (see Figure 27).

To enable program interruptions that may occur during execution of the interrupt handling routines, IHEWERR follows this method of handling program interruptions:

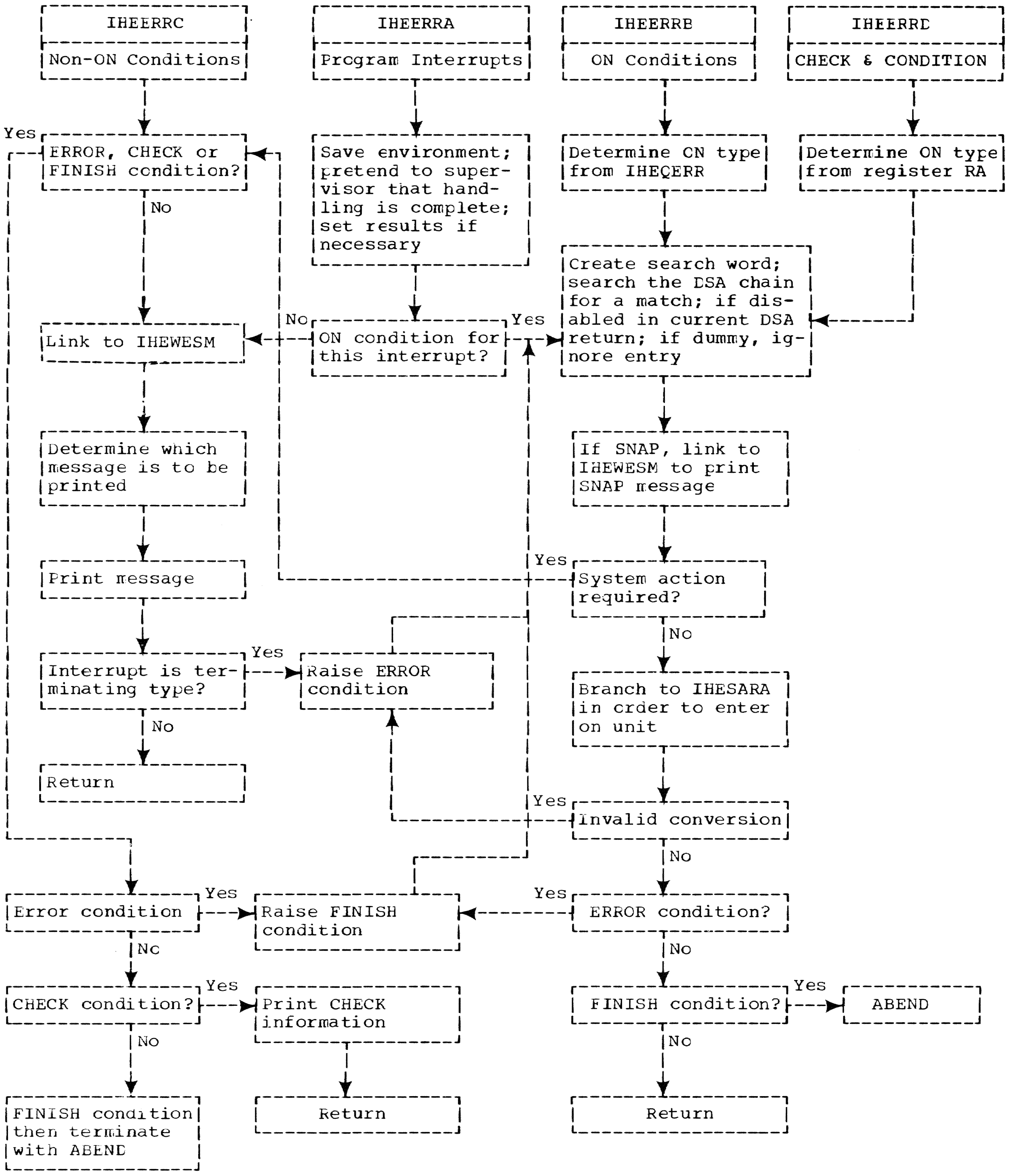

IHEERRC
Non-ON Conditions
IHEERRA
Program Interrupts
IHEERRB
ON Conditions
IHEERRD
CHECK & CONDITION
ERROR, CHECK or FINISH condition?
Yes
No
Save environment; pretend to supervisor that handling is complete; set results if necessary
Determine ON type from IHEQERR
Determine ON type from register RA
Create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

Figure 24. Flow Through the Error Handling Routine (IHEWERR)

Program Interruption		PL/I Condition
Interrupt Code	Meaning	
1	operation code	Execution error conditions not covered by a PL/I defined condition
2	privileged operation	
3	execution	
4	protection	
5	addressing	
6	specification	
7	data	
8	fixed-point overflow	FIXEDOVERFLOW
9	fixed-point divide	ZERO DIVIDE
10	decimal overflow	FIXEDOVERFLOW
11	decimal divide	ZERODIVIDE
12	exponent overflow	OVERFLOW
13	exponent underflow	UNDERFLOW
14	significance	not handled in PL/I
15	floating-point divide	ZERODIVIDE

Figure 25. Program Interruptions and PL/I Conditions

2. Bits 40 to 63 of the old VPSW in the 64 word entry are changed to contain the address of the appropriate entry point in IHE-WERR; control is returned to the supervisor.

3. TSS assumes the interruption has been handled satisfactorily and transfers control to the new address in the old VPSW; thus it enters module IHEWERR again.

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

When fixed-point or decimal overflow interruptions occur, the SIZE condition may be raised. Therefore when one of these interruptions occurs, the pseudo-register IHEQERR must be inspected to see if the SIZE code has been set. Similarly, if any of the divide interruptions 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 interruption.

Certain very unusual circumstances may result in a program interruption occurring during the execution of IHEWERR or of one of the library modules called from it. For example, if the program destroys the PRV, or the DSA chain, or parts of library work-space, then it is likely that sonner or later a specification or addressing interruption will occur.

Under these circumstances, to prevent any attempt to re-enter IHEERRA on account of the second interruption, SPEC and SIR macros are issued every time IHEWERR is entered. These macros provide that, in the event of an interruption, IHEWERR shall be entered at entry point IHEERRE. Similarly, a DIR macro is issued at each exit point, to restore IHEERRA as the normal entry point for program interruptions during the execution of compiled code and library routines.

When IHEERRE is entered, a message is printed on SYSOUT and the FINISH condition is raised.

ON CONDITIONS

The six classes of ON conditions defined in PL/I are shown in Figure 28. To deal satisfactorily with the situation when any of these conditions arise, IHEWERR 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.

	64 Word Entry	Word
	ID	1
	Forward Pointer to Next Entry	2
	Forward Page Pointer	3
Reg 0	Pointer to Interrupt Conditions	4
--->	Length = 120 Bytes	5
	Register 13	6
	Unused	7
	Register 14	8
	Register 15	9
	Register 0	10
	Register 1	11
	Registers 2-12	12-22
	Not used	23
	Not used	24
	Old VPSW	25-26
	Floating Point Register 0	27-28
	Floating Point Register 2	28-30
	Floating Point Register 4	31-32
	Floating Point Register 6	33-34
	Task Monitor RSPRV Flag	35
Reg 13	Pushdown Pointer from ISA	36
--->	Length = 108 Bytes	37
	Backward Link	38
	Forward Link	39
	Register 14 (Return Linkage)	40
	Register 15 (Entry Point)	41
	Registers 0-12	42-54
	PSECT Address of Called Program	55
	Available for Called Program	56-63
	Reserved	64

Figure 26. Information Available Upon Entry to an Interrupt Routine

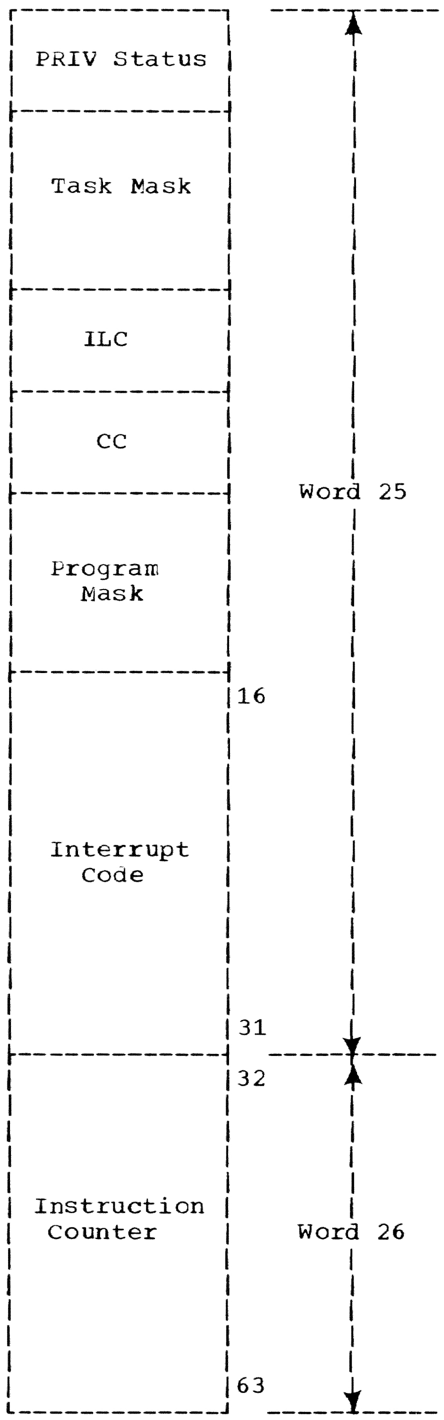


Figure 27. Old Virtual Program Status Word

In order to carry out these operations IHEWERR 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.

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 PENDING ENDPAGE KEY NAME RECORD TRANSMIT UNDEFINEDFILE	No	Always enabled
Program check-out	CHECK SUBSCRIPT-RANGE STRINGRANGE	Yes	Disabled
Programmer named	CONDITION	No	Always enabled
System action	ERROR #FINISH	No	Always enabled

Figure 28. PL/I ON Conditions

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 29.)

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)	Error code	A (Symbol name & length)
CHECK (variable)	Error code	A (Symbol table)
Others	Error code	Nothing stored

Figure 29. Format of the Search Word Comparator

2nd word: Byte 1: Bits 0, 1 and 4 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 disabled 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, IHEWERR gains control from one of the following:

1. TSS/360 (program interruptions)

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 29).

When the search word has been created, IHEWERR 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 interruption.

      Not disabled: The DSA is ignored.

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

   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 second word of 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 IHEWERR; the program returns to the point of interruption.

   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 Time Sharing System: PL/I Programmer's Guide.

   SNAP action not required: Proceed normally.

When a match has been 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 earlier, in 'PL/I Object Program Management'. In order to pass control to the on-unit, the recursion subroutine in IHEWSAP 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 IHEWERR, 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 'PL/I Object Program Management'.

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 IHEWERR. 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 explanations of their use are given in IBM System/360 Time Sharing System: PL/I Programmer's Guide.

MISCELLANEOUS TSS/360 INTERFACES

One function of the PL/I Library is to provide a standard interface with TSS/360 which can be utilized by compiled code. The implementation described here concerns support for PL/I language statements and functions with a TSS/360 interface that does not fall into one of the categories already discussed in this section. These are the PL/I statements DISPLAY, DELAY, STOP and EXIT, and the built-in functions TIME and DATE.

TSS/360 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
TIME	EBCDTIME
DATE	EBCDTIME
DISPLAY	GATWR, GATRD

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

DELAY
: the execution of the PL/I program is suspended for the required time.

EXIT and STOP
: both these statements raise the FINISH condition and then cause normal 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. (Since it is only possible to obtain the nearest hundreth under TSS/360, the last digit will always be zero.)

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 SYSOUT with no interruption in execution or, if a reply is expected, execution is suspended until the user's reply is received. If the EVENT option is employed, it will not have any effect on program execution; the program <u>will</u> not resume execution until a response has been received from the user's SYSIN.

DATA PROCESSING ROUTINES

I/O EDITING AND DATA CONVERSION

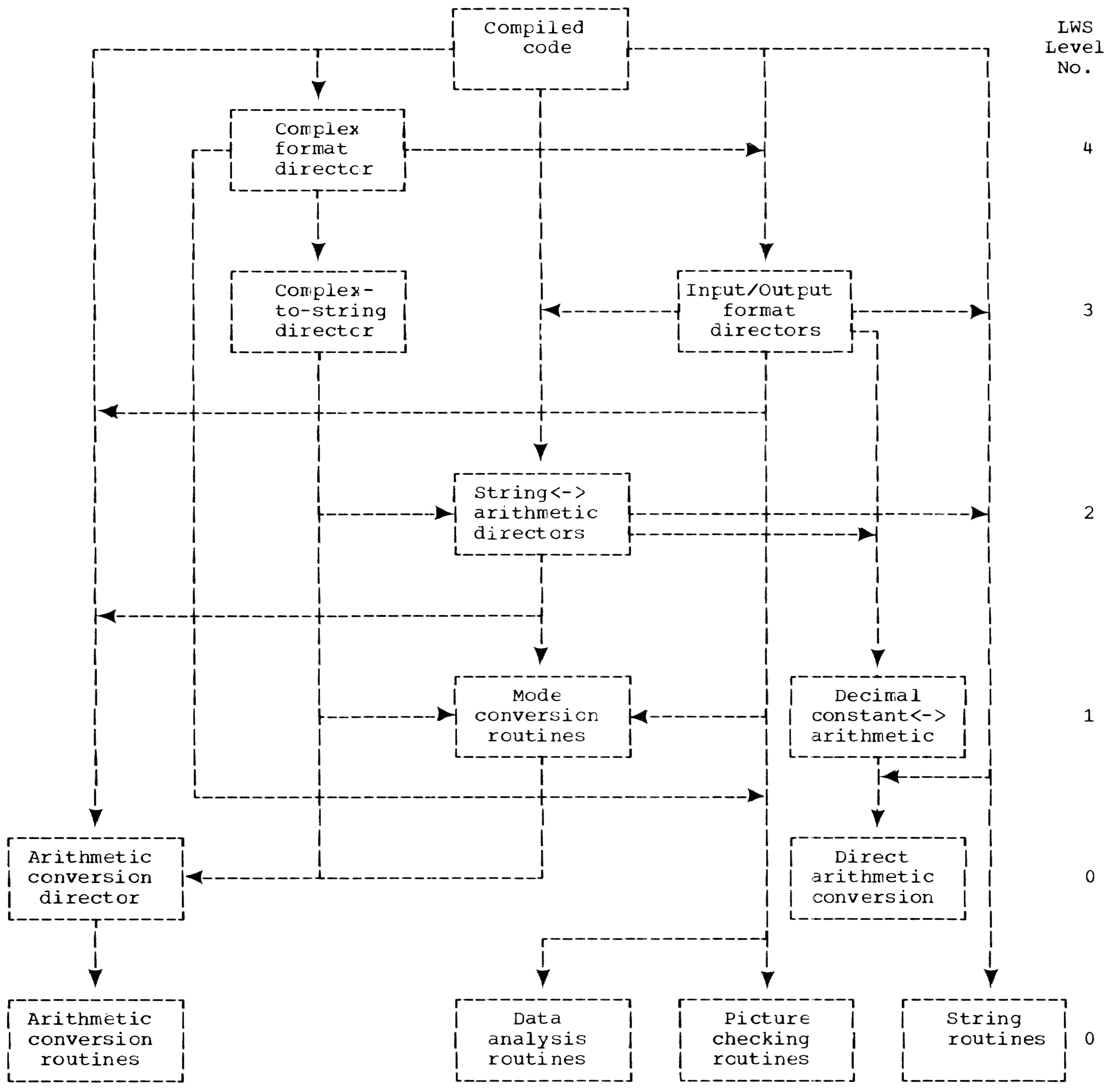
PL/I allows the user a wide choice in selecting the representation for his data, both on the external medium and internally in storage; considerable flexibility is permitted in specifying changes of data type and form. The library conversion package is designed to implement the full set of editing and 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 PL/I compiler analyzes 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.

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'). The exception is IHEWVCS (complex-to-string director) which is called by the system external standard calling sequence. The letters in the module name indicate the module usage; see Figure 30.

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

Figure 30. Module Usage indicated by Letters of Module Name


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 31. Structure of the Conversion Package

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 31.

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

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. The hexadecimal '10' superimposed on the DED flag byte indicates the presence of a halfword fixed point binary variable. Bit 3 is set to 1 and bit 6 is set to 0.

Figure 32. DED Flag Byte for Character Representation of an Arithmetic Data Item

control blocks are described in Section IV. The exception is that of a DED generated at execution time for communication between library modules (see Figure 32).

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 fulfill two main purposes:

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

2. The controling 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 (IHEWDMA), 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 35.) 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 director 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

PL/I Format Item	Director	Module Name	
		Input	Output
Complex	C	IHEWDIM	IHEWDOM
Fixed and floating point	F/E	IHEWDIA	IHEWDOA
Bit string	B	IHEWDID	IHEWDOD
Character string	A	IHEWDIB	IHEWDOB
Picture	P(DEC,STL)	IHEWDIE	IHEWDOE
	P(CHAR)	IHEWDIB	IHEWDOB

Figure 33. Input/Output Directors for PL/I Format Items

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 earlier by the execution-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 33.

I/O EDITING

Complex Directors: Complex format items on the external medium may have real and imaginary parts of different 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 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 IHEWIOD 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 pointed 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 31.)

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 IHEWERR, 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 34.

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 34. Conversion for List/Data Directed I/O

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 Time Sharing System: PL/I 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, shown in Figure 35, are used by list-, edit- and data-directed I/O and in some internal conversions.

	TO:		
	Arithmetic	String	
		Bit	Character
FROM: Arithmetic	-	IHEWDNB	IHEWDNC
Bit string	IHEWDBN	-	-
Character string	IHEWDCN	-	-

Figure 35. 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 36 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	IHEWVSA	IHEWVSB	IHEWVSF
Character	IHEWVSD	IHEWVSC	IHEWVSE

Figure 36. Modules for String Conversions

ARITHMETIC CONVERSIONS

A direct routine IHEWVQA 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.

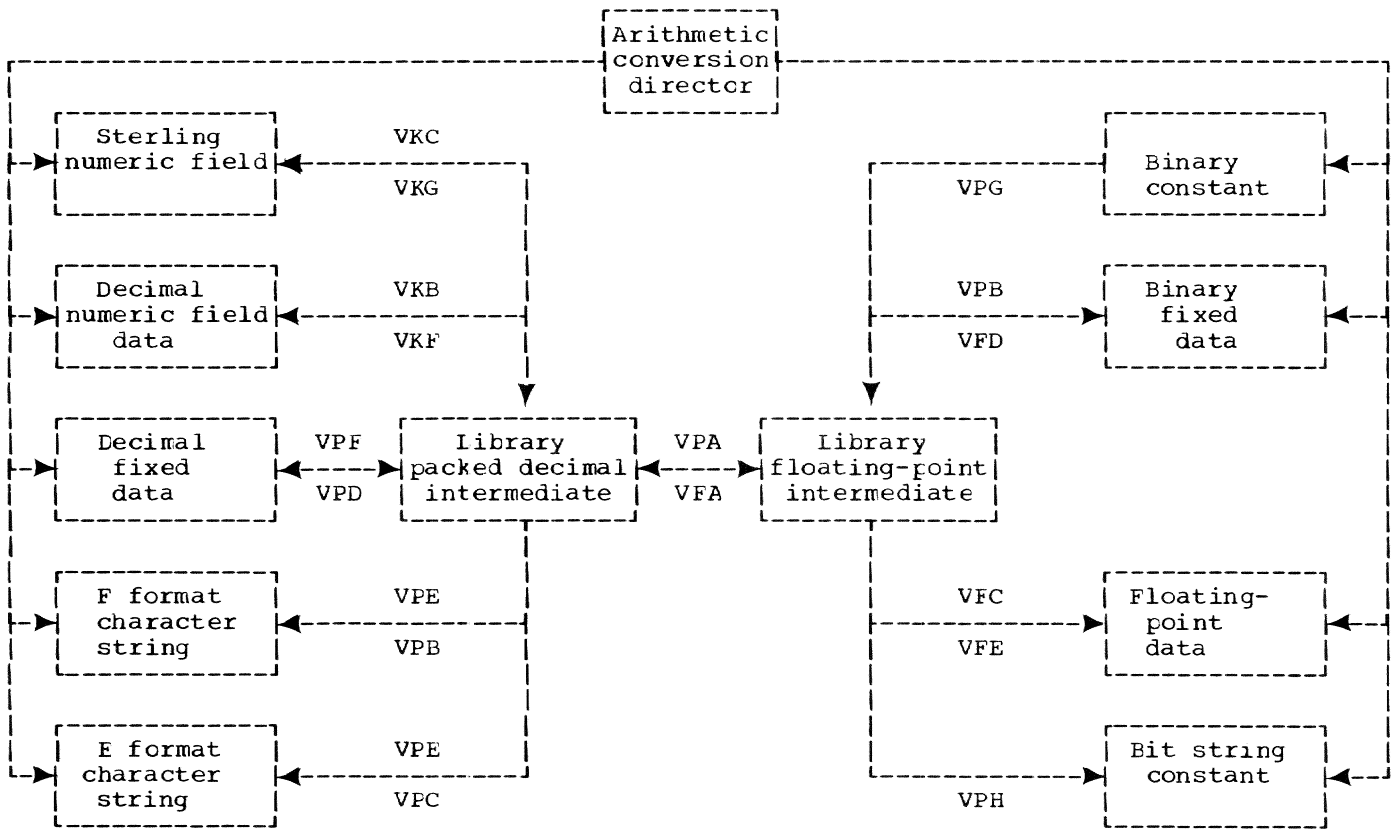
This package carries out editing and conversions for all type arithmetic source fields which have type arithmetic target fields. It also handles conversions of format items and constants, which are character representations of arithmetic type data. The flow control through this sub-package is achieved 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 37.



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 37. Structure of the Arithmetic Conversion Package

The arithmetic conversion director (IHE-WDMA) 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 and 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.

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 or 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 IHEWDIL (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 IHEWERR. 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)

The internal error codes passed to IHEWERR for ON-conditions are listed in Appendix E.

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 Time Sharing System: PL/I Reference Manual.

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 38.)

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.

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 38. Conversion Code Set in IHEQERR

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 Time Sharing System: PL/I 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 interruption may be ascertained in an ERROR unit and appropriate action may be taken. A list of the ONCODEs is given in IBM System/360 Time Sharing System: PL/I Reference Manual.

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 seven of the module name indicates the type of data concerned; the meanings of this character are given in Figure 39.

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 39. Relationship of Data Form and Seventh Character of Module Name

String Operations and Functions

The library string package contains modules for handling both bit and character string. The characteristics of the string operations and functions are listed in Figure 40. 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 PL/I standard for all functions except SUBSTR, INDEX and BOOL. 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 staisfied.

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 recognizing instances of byte-alignment which are suitable for the byte-aligned bit functions provided.

PL/I Operation	PL/I Function	Bit String		Character String
		General	Byte-aligned	
And	-	Use BOOL	IHEWBSA	-
Or	-	Use BOOL	IHEWBSO	-
Not	-	Use BOOL	IHEWBSN	-
Concatenate	REPEAT	IHEWBSK	-	IHEWCSK
Compare	-	IHEWBSD	IHEWBSC	IHEWCSC
Assign	-	IHEWBSK	IHEWBSM	IHEWCSM
Fill	-	IHEWBSM	-	IHEWCSM
-	HIGH/LOW	-	-	IHEWCSM
-	SUBSTR	IHEWBSS	-	IHEWCSS
-	INDEX	IHEWBSI	-	IHEWCSI
-	BOOL	IHEWBSF	-	-

Figure 40. String Operations and Functions

ARITHEMETIC OPERATIONS				
Operation	Binary fixed	Decimal fixed	Short float	Long float
Real Operations				
Integer exponentiation: x**n	IHEWXIB	IHEWXID	IHEWXIS	IHEWXIL
General exponentiation: x**y	-	-	IHEWXXS	IHEWXXL
Shift-and-assign, Shift-and-load	-	IHEWAPD	-	-
Complex Operations				
Multiplication/division: z_1*z_2, z_1/z_2	IHEWMZU	IHEWMZV	-	-
Multiplication: z_1*z_2	-	-	IHEWMZW	IHEWMZZ
Division: z_1/z_2	-	-	IHEWDZW	IHEWDZZ
Integer exponentiation: z**n	IHEWXIU	IHEWXIV	IHEWXIW	IHEWXIZ
General exponentiation: z_1**z_2	-	-	IHEWXXW	IHEWXXZ

Figure 41. Arithmetic Operations

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.

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 compiler neither generates in-line code nor (as for the functions FIXED, FLOAT, BINARY, AND DECIMAL) uses the library conversion package. (See Figures 41 and 42.)

Linkage between compiled code and the arithmetic modules is established by means of the system standard for the functions supported and by means of the PL/I standard for the operations 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 accomodate the result precision as defined by the language.

ARITHMETIC FUNCTIONS				
Function	Binary fixed	Decimal fixed	Short float	Long float
Real Arguments				
MAX, MIN	IHEWMXB	IHEWMXD	IHEWMXS	IHEWMXL
ADD	-	IHEWADD	-	-
Complex Arguments				
ADD	-	IHEWADV	-	-
MULTIPLY	IHEWMPU	IHEWMPV	-	-
DIVIDE	IHEWDVU	IHEWDVV	-	-
ABS	IHEWABU	IHEWABV	IHEWABW	IHEWABZ

Figure 42. Arithmetic Functions

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 execution-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 (see Figure 43).

Linkage to all mathematical subroutines is by means of the system standard.

Real Arguments		
Function	Short float	Long float
SQRT	IHEWSQS	IHEWSQL
EXP	IHEWEXS	IHEWEXL
LOG,LOG2,LOG10	IHEWLNS	IHEWLNL
SIN, COS,SIND,COSD	IHEWSNS	IHEWSNL
TAN, TAND	IHEWTNS	IHEWTNL
ATAN, ATAND	IHEWATS	IHEWATL
SINH, COSH	IHEWSHS	IHEWSHL
TANH	IHEWTHS	IHEWTHL
ATANH	IHEWHTS	IHEWHTL
ERF,ERFC	IHEWEFS	IHEWEFL

Complex Arguments		
Function	Short float	Long float
SQRT	IHEWSQW	IHEWSQZ
EXP	IHEWEXW	IHEWEXZ
LOG	IHEWLNW	IHEWLNZ
SIN,COS,SINH,COSH	IHEWSNW	IHEWSNZ
TAN, TANH	IHEWTNW	IHEWTNZ
ATAN, ATANH	IHEWATW	IHEWATZ

Figure 43. Mathematical Functions

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.

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. (See Figure 44.) Calls to array function modules are by means of the 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 virtual 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 Section IV. 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	IHEWJXS IHEWNL1	IHEWJXI IHEWNL2

Note: IHEWJXI 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	IHEWSSF	IHEWSMF	IHEWSSG	IHEWSMG	IHEWSSH	IHEWSMH
Complex	IHEWSSX	IHEWSMX	IHEWSSG	IHEWSMG	IHEWSSH	IHEWSMH
PROD real	IHEWPSF	IHEWPDF	IHEWPSS	IHEWPDS	IHEWPSL	IHEWPDL
complex	IHEWPSX	IHEWPDX	IHEWPSW	IHEWPDW	IHEWPSZ	IHEWPDZ
POLY real	IHEWYGF		IHEWYGS		IHEWYGL	
complex	IHEWYGX		IHEWYGW		IHEWYGZ	

Figure 44. Array Indexers and Functions

SECTION II

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 (IHE-WERR), no reference is made to this module in the 'Calls' section.

CONTROL PROGRAM INTERFACES

The 'Calls' and 'Called by' sections include the use of the CALL macro to pass control. In those cases mentioned in the section on OPEN processing, a direct branch is taken.

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 IHEWVSA, IHEWVSB, IHEWVSC, IHEWVSD, IHEWVSE, and IHEWVSF.

Strings: A source string field may coincide with a target string field in the modules listed in Figure 45. 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	Either Source Field
IHEWBSA	Yes	-
IHEWBSO	-	Yes
IHEWBSK	Yes	-
IHEWBSM	Yes	-
IHEWBSF	-	Yes
IHEWCSK	Yes	-
IHEWCSM	Yes	-

Figure 45. Coincidence of Source and Target Fields in Some String Modules

The first byte of the result produced by the comparison modules IHEWBSC, IHEWBSD, and IHEWCSC 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 IHEWXIU, IHEWXIV, IHEWXIW, IHEWXIZ, IHEWXXL and IHEWXXS.

Mathematical: Target fields may, if desired, be overlapped with source fields in all cases except IHEWEFL, IHEWEFS, IHEWLNW, IHEWLNZ, IHEWSQW and IHEWSQZ.

IHEWABU

Entry 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

IHEWABV

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

IHEWABW

Calls: IHEWSQS

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, IHEWSQW

IHEWABZ

Calls: IHEWSQL

Entry Point: IHEABZ0

Function: ABS(z), when z is complex floating-point.

Linkage:

```
   RA:  A(Parameter list)
   Parameter list:
     A(z)
     A(Target)
```

Called by: Compiled code, IHEWSQZ

IHEWADD

Calls: IHEWAPD

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, IHEWADV

IHEWADV

Calls: IHEWADD

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

IHEWAPD

Calls: IHEERRB

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: IHEWADD, IHEWDVV, IHEWMPV

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: IHEWADD, IHEWDVV

IHEWATL

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, IHEWATZ, IHEWLNZ

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

IHEWATS

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, IHEWATW, IHEWLNW

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

IHEWATW

Calls: IHEWATS, IHEWHTS

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

Calls: IHEATS2, IHEWHTS

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

IHEWATZ

Calls: IHEWATL, IHEWHTL

Entry Point: IHEATZN

Calls: IHEATL2, IHEWHTL

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

Calls: IHEATL2, IHEWHTL

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

IHEWBEG

Entry Point: IHEBEGN

Function: Issues a GATWR macro instruction if the program does not have a main procedure.

Linkage: None

Called by: Compiled code

IHEWBSA

Entry Point: IHEBSAO

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

IHEWBSC

Entry Point: IHEBSC0

Function: To compare two byte-alighed bit strings.

Linkage:

```
  RA:  A(SDV of first operand)
  RB:  A(SDV of second operand)
  RC:  A(Target)
```

Called by: Compiled code

IHEWBSD

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

IHEWBSF

Entry Point: IHEBSF0

Function: BOOL (Bit string, bit string, string n_1 n_2 n_3 n).

Linkage:

```
  RA:  A(Parameter list)
  Parameter list:
    A(SDV of first source string)
    A(SDV of second source string)
    A(Fullword containing bit pattern n₁,
```

n_2 n_3 n right justified)
A(SDV of target field)

Called by: Compiled code

IHEWBSI

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

IHEWBSK

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, IHESTGA

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

IHEWBSM

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

IHEWBSN

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

IHEWBSO

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

IHEWBSS

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

IHEWBST

Calls: IHEWBSF, IHEWBSI, IHEWBSS

Entry Point: IHEBSTA

Function: Translate bit string

Linkage:

```
   RA:  A(Parameter list)
   Parameter list:
     A(SOURCE/TARGET SDV)
     A(REPLACEMENT SDV)
     A(POSITIONAL SDV)
```

Called by: Compiled code

IHEWBSV

Entry Point: IHEBSVA

Function: Verify bit string

Linkage:

```
   RA:  A(Parameter list)
   Parameter list:
     A(E1 SDV)
     A(E2 SDV)
     A(Result field)
```

Called by: Compiled code

IHEWCFA

Entry Point: IHECFAA

Function: ONLOC: Locates the BCD name of the procedure that contains the PL/I interruption 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

IHEWCFB

Entry Point: IHECFBA

Function: ONCODE: Returns a value corresponding to the condition which caused the interruption. 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

IHEWCFC

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 for all TSS/360 applications.

IHEWCKP

Entry Points: IHECKPS, IHECKPT

Function: Issues diagnostic message if attempt is made to use CHECKPOINT facility, and continues execution.

Linkage: None

Called by: Compiled code (CALL IHECKPS or CALL IHECKPT)

IHEWCLT

Calls: IHEWSAP, Supervisor (CLOSE, DCBD, FREEMAIN)

Entry Point: IHECLTA

Function: Close files:

1. Free FCB.

2. Set file register to zero.

3. Remove file from IHEQFOP chain.

4. Purge outstanding I/O events, setting event variables complete, abnormal, and inactive.

No close is issued if the file is SYSIN or SYSOUT.

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: IHEWOCL

Entry Point: IHECLTB

Function: To close all files when a PL/I program 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 indi-
 cates end of parameter list.)
```

Called by: IHEWOCL

IHEWCNT

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)
```

Called by: Compiled code

Entry Point: IHECNTB

Function: Returns current line number (LINENO).

Linkage: As for IHECNTA

Called by: Compiled code

IHEWCSC

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

IHEWCSI

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

IHEWCSK

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

IHEWCSM

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: Complied 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: Complied code

IHEWCSS

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

IHEWCST

Entry Point: IHECSTA

Function: Supplements translate character string

Linkage:

```
RA:  A(Parameter list)
Parameter list:
  A(SDV of SOURCE/TARGET)
  A(SDV of REPLACEMENT)
  A(SDV of POSITIONAL)
  A(Translate table)
```

Called by: Compiled code

IHEWCSV

Entry Point: IHECSVA

Function: Supplements verify character string

Linkage:

```
RA:  A(Parameter list)
Parameter list:
  A(E1 SDV)
```

A(E2 SDV)
A(Translate table)
A(Result field)

Called by: Compiled code

IHEWCVC

Calls: Supervisor (DIR, EBCDTIME, FREE-POOL, GATWR, GETBUF, GETPOOL, PAUSE, SIR, SPEC, STIMER, XTRCT)

Function: This module is a table containing the non-shareable part of the PL/I library. It consists of V- and A-type address constants, L-form macros, and executable macros with parameter lists or address constants. Pseudo-register IHEQCTS contains the base address of the module.

Called by: Compiled code

IHEWDBN

Calls: IHEWDMA, IHEWUPA, IHEWUPB

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, IHEWDCA, IHEW-DOE, IHEWDOM

IHEWDCN

Calls: IHEWDMA, IHEWUPA, IHEWUPB, IHEWVQB

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, IHEWDIB, IHEW-DOA, IHEWDOE

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

IHEWDDI

Calls: IHEWDDJ, IHEWIOF, IHEWLDI, IHEWSAP

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 table$_1$)
.
.
.
A(Symbol table$_n$)
(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

IHEWDDJ

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: IHEDDIA

IHEWDDO

Calls: IHEWDDP, IHEWIOF, IHEWLDO, IHEWPRT

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 entry₁)
         .
         .
         .
     A(Symbol table entryₙ)
     (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 entry₁)
     A(Element address₁)
         .
         .
         .
     A(Symbol table entryₙ)
     A(Element addressₙ)
     (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: IHEWERR, IHESAPA

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

IHEWDDP

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: IHEWDDO

Entry Point: IHEDDPB

Function: To perform subscript output.

Linkage:

```
   RA:  A(Parameter list)
   Parameter list:  A(VDA)
```

Called by: IHEWDDO

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: IHEWDDO

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: IHEWDDO

IHEWDIA

Calls: IHEWDMA, IHEWDNB, IHEWDNC, IHEWIOD, IHEWUPA, IHEWUPB, IHEWVCA, IHEWVQB, IHEWV-SA, IHEWVSC

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, IHEWDIM

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

IHEWDIB

Calls: IHEWDCN, IHEWIOD, IHEWKCD, IHEWVSC, IHEWVSD, IHEWVSE

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

IHEWDID

Calls: IHEWDBN, IHEWDMA, IHEWIOD, IHEWUPA, IHEWUPB, IHEWVSC, IHEWVSD, IHEWVSE

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

IHEWDIE

Calls: IHEWDMA, IHEWDMB, IHEWDMC, IHEWIOD, IHEWKCA, IHEWKCB, IHEWUPA, IHEWUPB, IHEWVSC, IHEWVSD, IHEWVSE

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, IHEWDIM

IHEWDIL

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

IHEWDIM

Calls: IHEWDIA, IHEWDIE, IHEWIOD, IHEWKCA, IHEWVCA, IHEWVCS

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

IHEWDMA

Transfers Control to: IHEWVFD, IHEWVFE, IHEWVKB, IHEWVKC, IHEWVPE, IHEWVPF, IHEWVPG, IHEWVPH

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, IHEWDBN, IHEWDCN, IHEWDNB, IHEWDNC, IHEWLDI, IHEWPDF, IHEWPDX, IHEWPSF, IHEWPSX, IHEWSMF, IHEWSMX, IHEWSSF, IHEWSSX, IHEWUPB, IHEWVCS, IHEWVFA, IHEWVFB, IHEWVFC, IHEWVPA, IHEWVPB, IHEWVPC, IHEWVPD, IHEWVKF, IHEWVKG, IHEWYGF, IHEWYGX

IHEWDNB

Calls: IHEWDMA, IHEWVSA

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, IHEWDIA, IHEWDIE, IHEWDOD, IHEWVCS

IHEWDNC

Calls: IHEWDMA, IHEWUPA, IHEWVQC, IHEWVSC, IHEWVSE

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, IHEWDIA, IHEWDIE, IHEWDOA, IHEWDOB, IHEWLDI, IHEWLDO, IHEWVCS

IHEWDOA

Calls: IHEWDBN, IHEWDCN, IHEWDMA, IHEWIOD, IHEWVQC

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

Entry Point: IHEDOAB

Function: To direct the conversion of internal data to external E format.

Linkage: As for IHEDOAA

Called by: As for IHEDOAA

IHEWDOB

Calls: IHEWDNC, IHEWIOD, IHEWVSB, IHEWVSC, IHEWVSE, IHEWVSF

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

IHEWDOD

Calls: IHEWDNB, IHEWIOD, IHEWVSB, IHEWVSC

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

IHEWDOE

Calls: IHEWDBN, IHEWDCN, IHEWDMA, IHEWIOD, IHEWVSB

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, IHEWDOM

IHEWDOM

Calls: IHEWDBN, IHEWUPA, IHEWUPB, IHEWVCA, IHEWVCS

Entry Point: IHEDOMA

Function: To direct the conversion of an internal data type to external C format data.

Linkage:

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

IHEWDSP

Calls: System (GATWR, GATRD, GETMAIN, FREEMAIN)

Entry Point: IHEDSPA

Function: To write a message to SYSOUT, with or without a reply. The EVENT option can be used for a message with a reply, but it will have no effect on program execution.

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

IHEWDUM

Calls: IHEWZZC

Entry Points: IHEDUMC, IHEDUMJ

Function: To index the current areas and PAUSE if conversational or dump the current areas if nonconversational. Execution then continues at the next statement.

Linkage:

RA: A(Parameter list)
Parameter list:
A(Fullword binary integer, in range 0 through 255)

Called by: Compiled code
(CALL IHEDUMC or CALL IHEDUMJ)

Entry Points: IHEDUMP, IHEDUMT

Function: Same as IHEDUMC, except that the PL/I program is terminated.

Linkage: Same as IHEDUMC

Called by: Compiled code
(CALL IHEDUMP or CALL IHEDUMT)

IHEWDVU

Entry Point: IHEDVUQ

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

IHEWDVV

Calls: IHEWAPD

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

IHEWDZW

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

IHEWDZZ

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

IHEWEFL

Calls: IHEWEXI

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

IHEWEFS

Calls: IHEWEXS

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

IHEWERD

Function: Part of the error-handling routines, it contains the data-processing error messages, and when required is called by IHEWESM.

IHEWERE

Function: 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 called by IHEWESM.

IHEWERI

Function: Part of the error-handling routines, it contains the input/output error messages for non-ON conditions, and when required is called by IHEWESM.

IHEWERO

Function: Part of the error-handling routines, it contains the error messages for non-I/O ON conditions, and when required is called by IHEWESM.

IHEWERP

Function: Part of the error-handling routines, it contains the error messages for I/O ON conditions, and when required is called by IHEWESM.

IHEWERR

Calls: System (SIR, DIR, SPEC), IHEWDDO, IHEWESM,IHEWPRT, IHEWSAP

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 (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 IHEWERR or in modules that IHEWERR calls. A diagnostic message is issued and return made to command mode.

Linkage: None

Called by: System

IHEWESM

Calls: System, IHEWERD, IHEWERE, IHEWERI, IHEWERO, IHEWERP, IHEWPRT, IHEWSAP, IHEWTSA

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(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: IHEWERR

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: IHEWERR

IHEWEXL

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, IHEWEFL, IHEWEXZ, IHEWSHL, IHEWSNZ, IHEWTHL, IHEWXXL

IHEWEXS

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, IHEWEFS, IHEWEXW, IHEWSHS, IHEWSNW, IHEWTHS, IHEWXXS

IHEWEXW

Calls: IHEWEXS, IHEWSNS

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, IHEWXXW

IHEWEXZ

Calls: IHEWEXL, IHEWSNL

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, IHEWXXZ

IHEWHTL

Calls: IHEWLNL

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, IHEWATZ

IHEWHTS

Calls: IHEWLNS

Entry Point: IHEHTS0

Function: ATANH (x), where x is real short floating-point.

Linkage:

```
RA:  A(Parameter list)
Parameter list:
  A(x)
  A(Target)
```

Called by: Compiled code, IHEWATW

IHEWIOA

Calls: IHEWIOP, IHEWOCL

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

IHEWIOB

Calls: IHEWIOP, IHEWOCL, System (GTWRC)

Entry Point: IHEIOBA

Function: To initialize the PUT operation, and to check the file status:

1. Open
2. Transmit error
3. Invalid

To invoke GATE for GATE files.

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

To invoke GATE for GATE files.

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

To invoke GATE for GATE files.

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

To invoke GATE for GATE files.

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

To invoke GATE for GATE files.

Linkage: As for IHEIOBC

Called by: Compiled code

Entry Point: IHEIOBT

Function: To terminate the PUT operation.

Linkage: None

Called by: Compiled code

IHEWIOC

Calls: IHEWSAP, IHEWTSA

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

IHEWIOD

Calls: IHEWIOF, IHEWSAP, IHEWPRT, IHEWPTT, IHEWTSA

Entry Point: IHEIODG

Function: To obtain the next data field from the record buffer(s).

Linkage: Library communication area (WSDV)

Called by: Format directors, IHEWIOX

Entry Point: IHEIODP

Function: To obtain space for a data field in the record buffer(s).

Linkage: As for IHEIODG

Called by: Format directors, IHEWIOX

Entry Point: IHEIODT

Function: To terminate the data field request.

Linkage: As for IHEIODG

Called by: Format directors

IHEWIOF

Calls: Data management (QSAM, VSAM, SYSIN, GATWR)

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: IHEWDDI, IHEWDDO, IHEWIOD, IHEWIOP, IHEWIOX, IHEWLDI, IHEWLDO, IHEWOCL, IHEWPRT

IHEWION

Calls: IHEWITB, IHEWITD, IHEWITE, IHEWITG, IHEWITP, IHEWOCL

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.

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

IHEWIOP

Calls: IHEWIOF

Entry Point: IHEIOPA

Function: PAGE option/format

Linkage: No explicit parameters

Called by: Compiled code, IHEWIOB, IHEWIBT

Entry Point: IHEIOPB

Function: SKIP option/format

Linkage:

```
   RA:  A(FED)
   FED:  Halfword binary integer
```

Called by: Compiled code, IHEWIOA, IHEWIOB, IHEWIBT

Entry Point: IHEIOPC

Function: LINE option/format

Linkage: As for IHEIOPB

Called by: As for IHEIOPA

IHEWIOX

Calls: IHEWIOD, IHEWIOF

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

IHEWITB

Calls: Data management (BSAM), System (GETMAIN)

Entry Point: IHEITBA

Function: To provide the interface with BSAM for CONSECUTIVE data sets with the UNBUFFERED attribute.

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: IHEWION

IHEWITD

Calls: Data management (VISAM), System (GETMAIN), IHEWSAP, IHEWTSA

Entry Point: IHEITDA

Function: To provide the interface with VISAM for creating or accessing INDEXED data sets when opened for SEQUENTIAL access (format F records).

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: IHEWION

IHEWITE

Calls: Data management (VISAM), System (GETMAIN), IHEWSAP

Entry Point: IHEITEA

Function: To provide the interface with VISAM for accessing INDEXED data sets opened for DIRECT access (format F records).

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: IHEWION

IHEWITG

Calls: Data management (QSAM, VSAM)

Entry Point: IHEITGA

Function: To provide the interface with QSAM and VSAM 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: IHEWION

IHEWITM

Calls: Data management (VISAM), System (GETMAIN), IHEWSAP

Entry Point: IHEITMA

Function: To provide the interface with VISAM for accessing INDEXED data sets opened for DIRECT access (format V records).

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: IHEWION

IHEWITN

Calls: Data management (VISAM), System (GETMAIN), IHEWSAP, IHEWTSA

Entry Point: IHEITNA

Function: To provide the interface with VISAM for creating or accessing INDEXED data sets when opened for SEQUENTIAL access (format V records).

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: IHEWION

IHEWJXI

Calls: IHEWSAP, IHEWTSA

Entry Point: IHEJXII

Function: To initialize IHEWJXI 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: IHEWNL2, IHEWSTG

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: IHEWOSW, IHEWPDF, IHEWPDL, IHEWPDS, IHEWPDW, IHEWPDX, IHEWPDZ, IHEWSMF, IHEWSMG, IHEWSMH, IHEWSMX, IHEWSTG

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

IHEWJXS

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: IHEWPSF, IHEWPSL, IHEWPSS, IHEWPSW, IHEWPSX, IHEWPSZ, IHEWSSF, IHEWSSG, IHEWSSH, IHEWSSX, IHEWNL1

IHEWKCA

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: IHEWDIE, IHEWDIM

IHEWKCB

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: IHEWDIE

IHEWKCD

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: IHEWDIB

Entry Point: IHEKCDB

Function: As for IHEKCDA, but the ONSOURCE address is not stored.

Linkage: As for IHEKCDA

Called by: IHEWLDI

IHEWLDI

Calls: IHEWDCN, IHEWDMA, IHEWDNB, IHEWDNC, IHEWIOF, IHEWKCD, IHEWPRT, IHEWSAP, IHEWTSA, IHEWUPA, IHEWUPB, IHEWVCA, IHEWVCS, IHEWVSC, IHEWVSD

Entry Point: IHELIIA

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(Variable₁)
     A(DED₁)
         .
         .
         .
     A(Variableₙ)
     A(DEDₙ)
      (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: IHEWDDI

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: IHEWDDI

IHEWLDO

Calls: IHEWDNC, IHEWIOF, IHEWVSB, IHEWVSC

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(Variable₁)
     A(DED₁)
         .
         .
         .
     A(Variableₙ)
     A(DEDₙ)
      (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: IHEWDDO
IHEWLNL

Entry Point: IHELNLE

Function: LOG(x), where x is real long floating-point.

Linkage:

RA: A(Parmeter list)
Parameter list:
A(x)
A(Target)

Called by: Compiled code IHEWHTL, IHEWLNZ, IHEWXXL, IHEWXXZ

Entry Point: IHELNL2

Function: LOG(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

IHEWLNS

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, IHEWHTS, IHEWLW, IHEWXXS, IHEWXXW

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

IHEWLNW

Calls: IHEWATS, IHEWLNS

Entry Point: IHELNWO

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, IHEWXXW

IHEWLNZ

Calls: IHEWATL, IHEWLNL

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, IHEWXXZ

IHEWLSP

Calls: System (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

IHEWMPU

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(DED for w)
  A(z)
  A(DED for z)
  A(Target)
  A(DED for target)
```

Called by: Compiled code

IHEWMPV

Calls: IHEWAPD

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

IHEWMXB

Entry Point: IHEMXBX

Function: MAX(x_1,x_2,...,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,...,x_n), where x_1,x_2 and x_n are real fixed-point binary.

Linkage: As for IHEMXBX

Called by: Compiled code

IHEWMXD

Entry Point: IHEMXDX

Function: MAX(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...,x_n), where x_1,x_2 and x_n are real fixed-point decimal.

Linkage: As for IHEMXDX

Called by: Compiled code

IHEWMXL

Entry Point: IHEMXLX

Function: MAX(x_1,x_2,...,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,...,x_n), where x_1,x_2 and x_n are real long floating-point.

Linkage: As for IHEMXLX

Called by: Compiled code

IHEWMXS

Entry Point: IHEMXSX

Function: MAX(x_1,x_2,...,x_n), where x_1,x_2 and x_n are real short floating-point.

Calls: IHEWJXS

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: IHEMXSN

Function: MIN(x_1,x_2,...,x_n), where x_1,x_2 and x_n are real short floating-point.

Linkage: As for IHEMXSX

Called by: Compiled code

IHEWMZU

Entry Point: IHEMZUM

Function: z_1*z_2, where z_1 and z_2 are complex fixed-point binary.

Linkage:

```
   RA:  A(z1)
  *RB:  A(DED for z1)
   RC:  A(z2)
  *RD:  A(DED for z2)
   RE:  A(Target)
  *RF:  A(Target DED)
```

Called by: Compiled code, IHEWXIU

Entry Point: IHEMZUD

Function: z_1/z_2, where z_1 and z_2 are complex fixed-point binary.

Linkage:

```
   RA:  A(z1)
   RB:  A(DED for z1)
   RC:  A(z2)
  *RD:  A(DED for z2)
   RE:  A(Target)
  *RF:  A(Target DED)
```

Called by: Compiled code

IHEWMZV

Entry Point: IHEMZVM

Function: z_1*z_2, where z_1 and z_2 are complex fixed-point decimal.

Linkage:

```
   RA:  A(z1)
   RB:  A(DED for z1)
   RC:  A(z2)
   RD:  A(DED for z2)
   RE:  A(Target)
  *RF:  A(Target DED)
```

Called by: Compiled code, IHEWXIV

Entry Point: IHEMZVD

Function: z_1/z_2, where z_1 and z_2 are complex fixed-point decimal.

Linkage: As for IHEMZVM

Called by: Compiled code

IHEWMZW

Entry Point: IHEMZW0

Function: z_1*z_2, where z_1 and z_2 are complex short floating-point.

Linkage:

```
   RA:  A(z1)
   RB:  A(z2)
   RC:  A(Target)
```

Called by: Compiled code, IHEWXIW

IHEWMZZ

Entry Point: IHEMZZ0

Function: z_1*z_2, where z_1 and z_2 are complex long floating-point.

Linkage:

```
   RA:  A(z1)
   RB:  A(z2)
   RC:  A(Target)
```

Called by: Compiled code, IHEWXIZ

IHEWNL1

Calls: IHEWJXS

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

IHEWNL2

Calls: IHEWJXI

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(IHEBSAO) for ALL, or
     A(IHEBSOO) 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

IHEWOCL

Calls: System (DCBD, FREEMAIN),IHEWCLT, IHEWIOF, IHEWITI, IHEWOPN, IHEWSAP

Entry Point: IHEOCLA

Function: Explicit open: links to IHEOPNA; handles error conditions detected by IHEWOPN, IHEWOPO, IHEWOPP, IHEWOPQ or IHEWOPZ.

Linkage:

```
   RA:  A(OPEN parameter list)
   Parameter list:  See IHEOPN
```

Called by: Compiled code, IHEWPRT

Entry Point: IHEOCLB

Function: Explicit close: links to IHECITA.

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: IHEWIOB, IHEWION, IHEWSAP

Entry Point: IHEOCLD

Function: Implicit close:

1. When a PL/I program is terminated, to close all the files opened in the PL/I program (by linking to IHECLTB).

Linkage:

```
RA:  A(PRV of current task)
```

Called by: IHEWSAP

IHEWOPN

Calls: IHEWOPO (direct branch), IHEWOPZ, IHEWSAP, IHEWTSA

Entry Point: IHEOPNA

Function: Open files:

1. Merge declared attributes with OPEN options.

2. Invoke IHEWOPO.

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)
  (Reserve)
  A(LINESIZEn)/0
  A(PAGESIZEn)/0
  (High-order byte of last argument
  indicates end of parameter list.)
```

Called by: IHEWOCL

IHEWOPO

Calls: System (DCB, DCBD, GETMAIN), IHEWOPP (direct branch),IHEWSAP, IHEWTSA

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(IHEWOPN 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: IHEWOPN

IHEWOPP

Calls: System (DCBD, GETMAIN, OPEN), IHEWOPQ (direct branch), IHEWSAP, IHEWTSA

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(IHEWOPN 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: IHEWOPO

IHEWOPQ

Calls: System (DCBD, GETMAIN),IHEWSAP, IHEWTSA

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 VISAM linkage.
4. To simulate PUT PAGE when opening a PRINT file.

Linkage:

```
RA: A(Parameter list)
Parameter list:
  A(IHEWOPN 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 Code₁'
  AL3(DCLCB₁)
  XL4'Merged attributes₁'
      .
      .
      .
  X'Access/Organization Codeₙ'
  AL3(DCLCBₙ)
  XL4'Merged attributesₙ'

Data management OPEN parameter list:
  XL4'4*n' (where n is the number of
            files to be opened)
  X(Flags for data management OPEN
     executor₁)
  AL3(DCB₁)
      .
      .
      .
  X(Flags for data management OPEN
     executorₙ)
  AL3(DCBₙ)
```

Note: Access/Organization Code is described in the module listing.

Called by: IHEWOPP

IHEWOSD

Entry Point: IHEOSDA

Function: To obtain current date.

Linkage:

```
RA: A(Parameter list)
Parameter list: A(Target SDV)
```

Called by: Compiled code

IHEWOSE

Calls: IHEWSAP, IHEWTSA (to terminate the program and return to command mode)

Entry Point: IHEOSEA

Function: To terminate the PL/I program abnormally, raising the FINISH condition.

Called by: Compiled code

IHEWOSI

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

IHEWOSS

Calls: IHEWSAP, IHEWTSA (to terminate the program)

Entry Point: IHEOSSA

Function: To raise the FINISH condition and abnormally terminate the program.

Linkage: None

Called by: Compiled code

IHEWOST

Calls: EBCDTIME macro

Entry Point: IHEOSTA

Function: To use the EBCDTIME macro to obtain the time of day.

Linkage:

RA: A(Parameter list)
Parameter list: A(Target SDV)

Called by: Compiled code

IHEWOSW

Calls: System (FREEMAIN), IHEJXI, IHEWSAP, and the 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 to branch to the I/O transmit module (which raises I/O conditions if necessary).

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

IHEWPDF

Calls: IHEWDMA, IHEWJXI

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

IHEWPDL

Calls: IHEWJXI

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

IHEWPDS

Calls: IHEWJXI

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

IHEWPDW

Calls: IHEWJXI

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

IHEWPDX

Calls: IHEWDMA, IHEWJXI

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

IHEWPDZ

Calls: IHEWJXI

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

IHEWPRT

Calls: IHEWIOF, IHEWOCL, IHEWSAP

Entry Point: IHEPRTA

Function: To COPY a data field on SYSOUT.

Linkage:

```
RA:  A(Character string)
RB:  A(Halfword containing length of
       character string)
```

Called by: IHEWIOD, IHEWLDI

Entry Point: IHEPRTB

Function: To write an error message on SYSOUT. Also, to prepare for system action for CHECK condition.

Linkage: As for IHEPRTA

Called by: IHEWDDO, IHEWERR, IHEWESM, IHEWESS

IHEWPSF

Calls: IHEWDMA, IHEWJXS

Entry Point: IHEPSF0

Function: PROD for a single 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

IHEWPSL

Calls: IHEWJXS

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

IHEWPSS

Calls: IHEWJXS

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

IHEWPSW

Calls: IHEWJXS

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

IHEWPSX

Calls: IHEWDMA, IHEWJXS

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

IHEWPSZ

Calls: IHEWJXS

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

IHEWRES

Calls: IHESAFA

Entry Point: IHEREST

Function: To issue a diagnostic and call IHESAFA to raise FINISH if an attempt is made to use RESTART.

Linkage: None

Called by: Compiled code

Entry Point: IHERESN

Function: To issue diagnostic and return to user program if attempt is made to cancel automatic restart.

Linkage: None

Called by: Compiled code

IHEWSAP

Calls: System (FREEMAIN, GETMAIN, SPEC, SIR, DIR), IHEWBEG, IHEWMAN, IHEWDDO, IHEWOCL, IHEWPRT

Function: Storage management.

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 (zero
             bit of word 2 = 0) or PR off-
             set (zero 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 assocaited pseudo-register.

Linkage:

  RA: A(Controlled variable pseudo-register)

Called by: Compiled code

Entry Point: IHESAFQ

Function: To issue a DIR macro, close all files and return to the command mode.

Linkage: None

Called by: Library modules, IHEDUMP, IHEWOSE, IHEWOSS

Entry Point: IHESAPA

Function:

1. To provide a PRV and LWS for a main procedure, and to issue a SIR macro referencing the ICB created by a SPEC macro; then to transfer control to an address constant named IHEMAIN.

2. To pass a parameter from the statement invoking the PL/I program.

<u>Linkage</u>:

    L(LWS) from assembly of IHELIB

<u>Called by</u>: Initial entry

<u>Entry Point</u>: IHESAPB

<u>Function</u>: As for IHESAPA, except that the code handling the parameter is bypassed.

<u>Linkage</u>:

    L(LWS) from assembly of IHELIB

<u>Entry Point</u>: IHESAPC

<u>Function</u>: As for IHESAPA, but also reserves a 512-byte area for optimization purposes.

<u>Linkage</u>:

    L(LWS) from assembly of IHELIB

<u>Entry Point</u>: IHESAPD

<u>Function</u>: As for IHESAPB, but also reserves a 512-byte area for optimization purposes.

<u>Linkage</u>:

    L(LWS) from assembly or IHELIB

<u>Entry Point</u>: IHESARA

<u>Function</u>: 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.

<u>Linkage</u>:

```
RA:  A(Parameter list)
Parameter list:
  A(Entry parameter).  The entry parame-
   ter 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.
```

<u>Called by</u>: Compiled code, IHEWERR

<u>Entry Point</u>: IHESARC

<u>Function</u>: To place the return code in the pseudo-register IHEQRTC.

<u>Linkage</u>:

```
RA:  A(Parameter list)
Parameter list:
  A(Return code) (The return code is
   fixed binary with default precision.)
```

<u>Called by</u>: Compiled code

<u>IHEWSHL</u>

<u>Calls</u>: IHEWEXL

<u>Entry Point</u>: IHESHLS

<u>Function</u>: SINH(x), where x is real long floating-point.

<u>Linkage</u>:

```
RA:  A(Parameter list)
Parameter list:
  A(x)
  A(Target)
```

<u>Called by</u>: Compiled code

<u>Entry Point</u>: IHESHLC

<u>Function</u>: COSH(x), where x is real long floating-point.

<u>Linkage</u>: As for IHESHLS

<u>Called by</u>: Compiled code

<u>IHEWSHS</u>

<u>Calls</u>: IHEWEXS

<u>Entry Point</u>: IHESHSS

<u>Function</u>: SINH(x), where x is real short floating-point.

<u>Linkage</u>:

```
RA:  A(Parameter list)
Parameter list:
  A(x)
  A(Target)
```

<u>Called by</u>: Compiled code

<u>Entry Point</u>: IHESHSC

<u>Function</u>: COSH(x), where x is real short floating-point.

Linkage: As for IHESHSS

Called by: Compiled code

IHEWSMF

Calls: IHEWDMA, IHEWJXI

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

IHEWSMG

Calls: IHEWJXI

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

IHEWSMH

Calls: IHEWJXI

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

IHEWSMX

Calls: IHEWDMA, IHEWJXI

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

IHEWSNL

Entry Point: IHESNLS

Function: SIN(x), where x is real long floating-point.

Linkage:

```
   RA:  (Parameter list)
   Parameter list:
   A(x)
   A(Target)
```

Called by: Compiled code, IHEWEXZ, IHEWSNZ

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, IHEWEXZ, IHEWSNZ

Entry Point: IHESNLK

Function: COSD(x), where x is real long floating-point.

Linkage: As for IHESNLS

Called by: Compiled code

IHEWSNS

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, IHEWEXW, IHEWSNW

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, IHEWEXW, IHEWSNW

Entry Point: IHESNSK

Function: COSD(x), where x is real short floating-point.

Linkage: As for IHESNSS

Called by: Compiled code

IHEWSNW

Calls: IHEWEXS, IHEWSNS

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

IHEWSNZ

Calls: IHEWEXL, IHEWSNL

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

IHEWSPR

Entry Point: IHESPRT

Function: Contains the default DCLCB for SYSPRINT. This module is used only when no other DCLCB is provided.

Called by: IHEWPRT

IHEWSQL

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, IHEWABZ, IHEWSQZ

IHEWSQS

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, IHEWABW, IHEWSQW

IHEWSQW

Calls: IHEWSQS, IHEWABW

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

IHEWSQZ

Calls: IHEWABZ, IHEWSQL

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

IHEWSRC

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)
```

Called by: Compiled code

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)
```

Called by: Compiled code

Entry Point: IHESRCC

Function: Returns SDV of erroneous field (DATAFIELD). If used out of context, a null string is returned.

Linkage: As for IHESRCA

Called by: Compiled code

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

Called by: Compiled code

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

Called by: Compiled code

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

Called by: Compiled code

IHEWSRD

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)
```

Called by: Compiled code

IHEWSRT

Entry Points: IHEWSRTA, IHEWSRTB, IHEWSRTC, IHEWSRTD

Calls: GATWR macro

Function: To issue a diagnostic and return to command mode if an attempt is made to use the SORT/MERGE facility.

Called by: Compiled code

IHEWSSF

Calls: IHEWDMA, IHEWJXS

Entry Point: IHESSFO

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

IHEWSSG

Calls: IHEWJXS

Entry Point: IHESSGR

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

Entry Point: IHESSGC

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

IHEWSSH

Calls: IHEWJXS

Entry Point: IHESSHR

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

Entry Point: IHESSHC

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

IHEWSSX

Calls: IHEWDMA, IHEWJXS

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

IHEWSTG

Calls: IHEWJXI, IHEWBSK

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

IHEWSTP

Calls: IHEWBSK, IHEWBSM, IHEWJXI

Entry Point: IHESTPA

Function: Assigns a bit or character string to the elements of a scalar, array or structure variable.

Linkage:

RA: A(Dope Vector)
RB: A(Dope Vector Descriptor)
RC: A(SDV)

Called by: Compiled code

IHEWSTR

Calls: IHEWSAP, IHEWTSA

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

IHEWTAB

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.

IHEWTEA

Entry Point: IHETEAA

Function: Event variable assignment

Linkage:

RA: A(Source event variable)
RB: A(Target event variable)

Called by: Compiled code

IHEWTEV

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

IHEWTHL

Calls: IHEWEXL

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, IHEWTNZ

IHEWTHS

Calls: IHEWEXS

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, IHEWTNW

IHEWTNL

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, IHEWTNZ

Entry Point: IHETNLD

Function: TAND(x), where x is real long floating point.

Linkage: As for IHETNLR

Called by: Compiled code

IHEWTNS

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, IHEWTNW

Entry Point: IHETNSD

Function: TAND(x), where x is real short floating-point.

Linkage: As for IHETNSR

Called by: Compiled code

IHEWTNW

Calls: IHEWTHS, IHEWTNS

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

IHEWTNZ

Calls: IHEWTHL, IHEWTNL

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

IHEWTOM

Calls: Supervisor (GATWR)

Entry Point: IHETOMA

Function: Write error messages on SYSOUT.

Linkage:

```
RA:  Address of fullword containing the
     length of the message.
RB:  Address of the error message text.
```

Called by: IHEBEG, IHECKP, IHEOCL, IHERES, IHESRT.

IHEWTSA

Calls: System (GATWR)

Entry Points: IHETSAA; IHETSAP

Function: To issue a diagnostic and return to command mode because of an attempt to use multitasking.

IHEWUPA

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: IHEWDCN, IHEWDBN

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: IHEWDBN, IHEWDCN, IHEWDIA, IHEWLDI, IHEWDID, IHEWDIE, IHEWDNC, IHEWDOM, IHEWVCS

IHEWUPB

Calls: IHEWDMA

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: IHEWDCN, IHEWDBN

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: IHEWDBN, IHEWDCN, IHEWDIA, IHEWDID, IHEWDIE, IHEWDOM, IHEWLDI, IHEWVCS

IHEWVCA

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: IHEWDIA, IHEWDIM, IHEWDOM, IHEWLDI

IHEWVCS

Calls: IHEWDMA, IHEWDNB, IHEWDNC, IHEWUPA

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: IHEWDIM, IHEWDOM, IHEWLDI

Entry Point: IHEVCSB

Function: As for IHEVCSA but the conversion is to coded complex only.

Linkage: As for IHEVCSA

Called by: As for IHEVCSA

IHEWVFA

Calls: IHEWVTB

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: IHEWDMA

IHEWVFB

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: IHEWDMA

IHEWVFC

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: IHEWDMA

IHEWVFD

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: IHEWDMA

IHEWVFE

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: IHEWDMA

IHEWVKB

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: IHEWDMA

IHEWVKC

Entry Point: IHEVKCA

Function: To convert a sterling numeric field to packed decimal intermediate.

Linkage:

RA:	A(Source)
RB:	A(Source DED)

Called by: IHEWDMA

IHEWVKF

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: IHEWDMA

IHEWVKG

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: IHEWDMA

IHEWVPA

Calls: IHEWVTB

Entry Point: IHEVPAA

Function: Radix conversion: decimal to binary. To convert packed decimal intermediate to long precision floating-point.

Linkage:

WINT:	Decimal integer
WSCF:	Scale factor

Called by: IHEWDMA

IHEWVPB

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: IHEWDMA

IHEWVPC

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: IHEWDMA

IHEWVPD

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: IHEWDMA

IHEWVPE

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: IHEWDMA

IHEWVPF

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: IHEWDMA

IHEWVPG

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: IHEWDMA

IHEWVPH

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: IHEWDMA

IHEWVQA

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 SDV)
RB:	A(Source DED)
RC:	A(Target SDV)
RD:	A(Target DED)

Called by: Compiled code, IHEWVQB

IHEWVQB

Calls: IHEWVQA, IHEWVTB

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: IHEWDCN, IHEWDIA

IHEWVQC

Calls: IHEWVSC, IHEWVSE

Entry Point: IHEVQCA

Function: To convert some coded arithmetic data types to F or E format or character string.

Linkage:

RA:	A(Source SDV)
RB:	A(Source DED)
RC:	A(Target SDV)
RD:	A(Target DED)
WFDT:	A(FED)
WSWB:	Switches specifying type of target string

Called by: IHEWDNC, IHEWDOA

IHEWVSA

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, IHEWDIA, IHEWDNB

IHEWVSB

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, IHEWDOB, IHEW-DOD, IHEWDOE, IHEWLDO

IHEWVSC

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, IHEWDIA, IHEWDIB, IHEWDID, IHEWDIE, IHEWDNC, IHEWDOB, IHEWDOD, IHEWLDI, IHEWVQC

IHEWVSD

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, IHEWDIB, IHEWDID, IHEWDIE, IHEWLDI

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

IHEWVSE

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, IHEWDIB, IHEWDID, IHEWDIE, IHEWDOB

Entry Point: IHEVSEB

Function: As for IHEVSEA, but the ONSOURCE address is not stored.

Linkage: As for IHEVSEA, but without WODF

Called by: IHEWDNC, IHEWVQC

IHEWVSF

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, IHEWDOB

IHEWVTB

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 IHEVWQB radix conversion routines IHEWVPA, and IHEWVFA.

Linkage: Not called. Referenced as external data by IHEWVPA, IHEWVQB and IHEWVFA.

IHEWXIB

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

IHEWXID

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

IHEWXIL

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

IHEWXIS

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

IHEWXIU

Calls: IHEWMZU

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

IHEWXIV

Calls: IHEWMZV

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

IHEWXIW

Calls: IHEWMZW

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

IHEWXIZ

Calls: IHEWMZZ

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

IHEWXXL

Calls: IHEWEXL, IHEWLNL

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

IHEWXXS

Calls: IHEWEXS, IHEWLNS

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

IHEWXXW

Calls: IHEWEXW, IHEWLNS, IHEWLNW

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

IHEWXXZ

Calls: IHEWEXZ, IHEWLNL, IHEWLNZ

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

IHEWYGF

Calls: IHEWDMA

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

IHEWYGL

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

IHEWYGS

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

IHEWYGW

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 1)
     A(Argument 2)
     A(Target)
```

Called by: Compiled code

IHEWYGX

Calls: IHEWDMA

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

IHEWYGZ

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 argument2)
     A(Target)
```

Called by: Compiled code

IHEWZZC

Calls: IHEWZZF

Entry Point: IHEZZCA

Function: To provide information about open files and save areas. In nonconversational mode, a dump is given.

Linkage:

RA: A(Parameter list)
See source listing for parameter list.

Called by: IHEWDUM

IHEWZZF

Entry Point: IHEZZFA

Function: To provide the save area trace that forms part of the output produced by IHEWZZC.

Linkage:

RA: A(Parameter list)
See source listing for parameter list.

Called by: IHEWZZC

SECTION III

DATA AREA LAYOUTS

This is a description of all the compiler-generated control blocks used by the PL/I Library. They are in alphabetical order, except for the Input/Output Control Blocks and the Storage Management Control Blocks, which are described separately later in this section. All offsets are given in hexadecimal form.

ARRAY DOPE VECTOR (ADV)

This control block (see Figure 46) 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.

```
 0  2 3    7 8      15 16                   31
┌────┬─────┬─────────────────────────────────┐
│BtO │     │         Virtual origin          │
├────┴─────┴─────────────────────────────────┤
│               Multiplier1                  │
├────────────────────────────────────────────┤
│                    .                       │
│                    .                       │
│                    .                       │
├────────────────────────────────────────────┤
│               Multipliern                  │
├────────────────────┬───────────────────────┤
│  Upper bound1      │    Lower bound1       │
├────────────────────┼───────────────────────┤
│         .          │          .            │
│         .          │          .            │
│         .          │          .            │
├────────────────────┼───────────────────────┤
│  Upper boundn      │    Lower boundn       │
└────────────────────┴───────────────────────┘
```

Figure 46. Format of the Array Dope Vector (ADV)

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.

Definitions of ADV Fields:

BtO (= 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.

ADV Algorithm
: given subscript values for an n-dimensional array, the address of any element is computed as:

```
                         n
     Address = origin +      S *M
                        i=1

     where S = value of the ith subscript
           M = 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 virtual 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 47. Format of the Data Element Descriptor (DED)

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 47.

Definitions of DED fields:

Flags
: an eight-bit encoded form of declared information (Figure 48). 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(±1) 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.

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.

Note: The hexadecimal '10' superimposed on the DED Flag byte indicates the presence of a halfword fixed point binary variable. Bit 3 is set to 1 and bit 6 is set to 0.

Figure 48. Format of the DED Flag 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.

LIBRARY COMMUNICATION AREA (LCA)

The library communication area (LCA -- see Figure 49) is part of library workspace (LWS), the format of which is given in Figure 50. The use of LWS and LCA is described in "Communication Conventions" in Section II.

LIBRARY WORKSPACE (LWS)

The use of the Library Workspace (LWS) is described in Section II. The format of the LCA is given in Figure 49 and that of the SSA in Figure 51.

	Symbolic name	Length (bytes)	Function
0	WBR1	4	2nd address for communication in arithmetic conversion package.
4	WBR2	4	3rd 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 49. Library Communication Area (LCA)

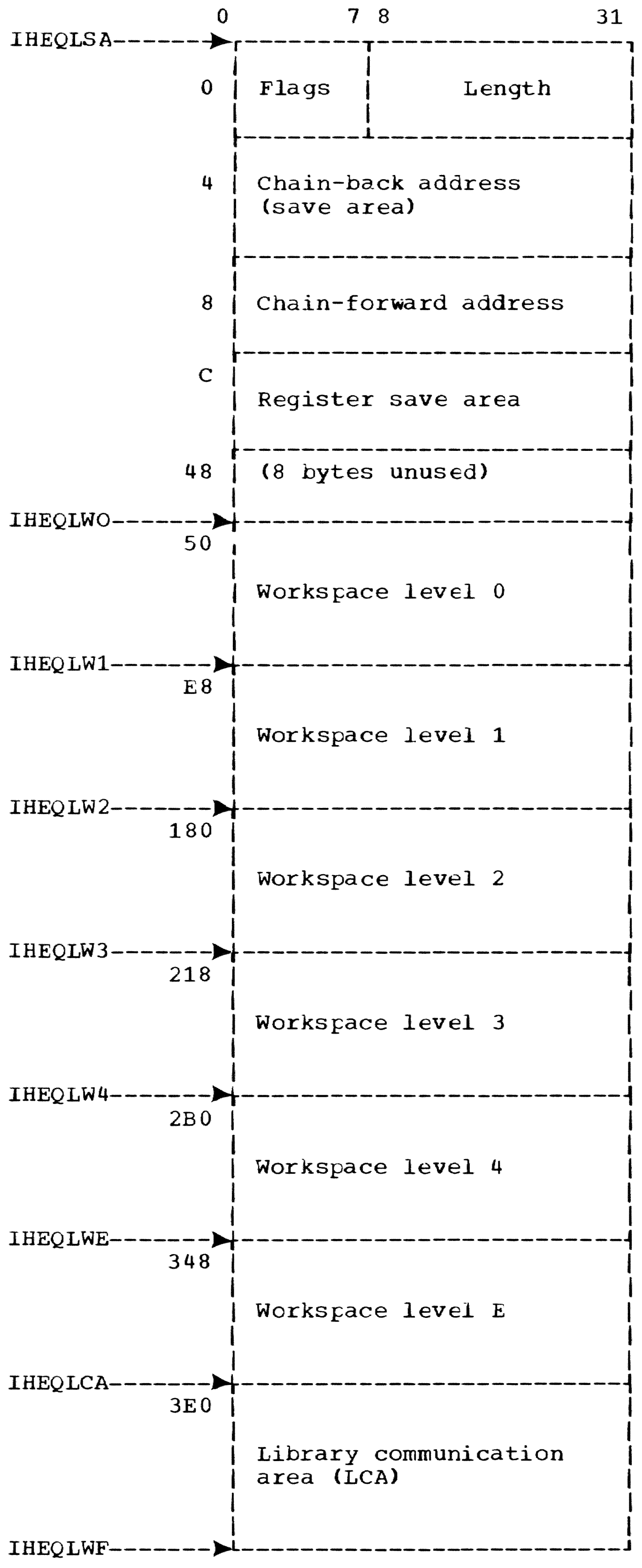


Figure 50. Standard Format of Library Workspace (LWS)

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 52.)

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.

Locations 14 through 48
    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. The register PR field is set by the subroutine in IHEWSAP that initializes the main procedure; it remains unchanged throughout the task. For some I/O macro expansions, the content of register 14 is stored at location 48.

STRING ARRAY DOPE VECTOR (SADV)

This control block (see Figure 53) 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.

Offset		General Register		Standard Save Area
Value	Symbolic Name		Symbolic Name	Usage (0 – 7 8 – 31)
0	OFCD	-	-	Flags (0–7) | Length (8–31)
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
48	-	14	LR,RY	

Figure 51. Format of the Standard Save Area (SSA)

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.

STRING DOPE VECTOR (SDV)

A string dope vector (SDV) is an 8-byte word-aligned block that specifies storage requirements for string data. The format of the SDV is shown in Figure 54.

<u>Definition of SDV fields:</u>

BtO (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.

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 conditionsions
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 52. Format of the SSA Flag Byte

```
 0                 15 16                  31
┌─────────────────────────────────────────┐
│                                         │
│                                         │
│                  ADV                    │
│                                         │
│                                         │
├────────────────────┬────────────────────┤
│  Maximum length    │  Current length/0  │
└────────────────────┴────────────────────┘
```

Figure 53. Format of the Primary String Array Dope Vector (SADV)

```
 0  2 3   7 8     15 16                  31
┌────┬─────┬──────────────────────────────┐
│BtO │     │   Byte address of string     │
├────┴─────┴──────────┬───────────────────┤
│  Maximum length     │  Current length   │
└─────────────────────┴───────────────────┘
```

Figure 54. String Dope Vector Format Vec

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.

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. Figure 55 describes the format of a SYMTAB entry.

```
 0       7 8      15  16                  31
┌─────────┬───────────────────────────────┐
│    0    │    Chain-forward address      │
├─────────┼───────────────────────────────┤
│ Length  │                               │
├─────────┘                               │
│              Identifier                 │
│                                         │
├─────────┬───────────────────────────────┤
│    D    │           A(DED)              │
├─────────┼───────────────────────────────┤
│  Flags  │           Field A             │
├─────────┴──────────┬────────────────────┤
│      Field B       │                    │
└────────────────────┴────────────────────┘
```

Figure 55. Format of the Symbol Table (SYMTAB)

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 Section II for description of storage class implementation and for definition of DSA.

This describes 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 Section II. An example of the chaining of I/O control blocks is included. In the diagrams, all offsets are in hexadecimal.

DECLARE CONTROL BLOCK (DCLCB)

The declare control block, shown in Figure 56, contains these fields:

DPRO
: halfword binary integer (set by the linkage editor) specifying the offset, within 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)*
xxxx	0101	TELEPROCESSING*

```
       0        7 8       15 16     23 24       31
       ┌──────────────────┬──────────────────┐
    0  │       DPRO       │       DCLA       │
       ├──────────────────┼──────────────────┤
    4  │       DBLK       │       DLRL       │
       ├─────────┬────────┼─────────┬────────┤
    8  │  DCLD   │  DBNO  │  DCLB   │  DCLC  │
       ├─────────┴────────┼─────────┴────────┤
    C  │       DXAL       │NCP Value Reserved│
       ├──────────────────┴──────────────────┤
   10  │             (Reserved)              │
       ├─────────────────────────────────────┤
   14  │             (Reserved)              │
       ├─────────┬───────────────────────────┤
   18  │  DFLN   │                           │
       ├─────────┘                           │
       │                                     │
       │                DFIL                 │
       │                                     │
       │                                     │
       │                                     │
       └─────────────────────────────────────┘
```

Figure 56. Format of the Declare Control Block (DCLCB)

(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.

TP
: maximum message length.

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.

----

*Not used in TSS/360

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	TRANSIENT*
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/G
6	1	R
7	-	(Reserved)

DXAL
halfword binary integer specifying the count in the INDEXAREA area enviornment 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 DDEF 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.)

*Not used in TSS/360.

EVENT VARIABLE

The format of this control block is shown in Figure 57; 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 event chain, which is anchored in the pseudo-register IHEQEVT, and includes all active I/O event variables associated with the PL/I program. This chain facilitates the setting of those event variables that are not being waited on inactive, complete, and abnormal on termination of the PL/I program.

An example of the chaining of event variables is given at the end of this section on Input/Output Control Blocks.

EVF1
8-bit code containing implementation flags:

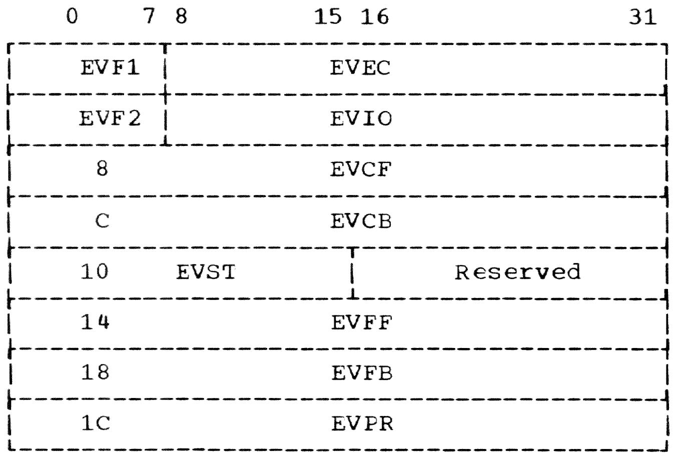


Figure 57. Format of the Event Variable

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 the first word	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	EMWE
Complete	0100 0000	EMCP

EVIO
: not used.

EVCF
: event variable chain-forward pointer.*

EVCB
: event variable chain-back pointer.*

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 PL/I program.

FILE CONTROL BLOCK (FCB)

The format of the file control block is determined by whether the file is stream-oriented (Figure 58) or record-oriented (Figure 59). The fields in the FCB are:

----
*Not used in TSS/360.

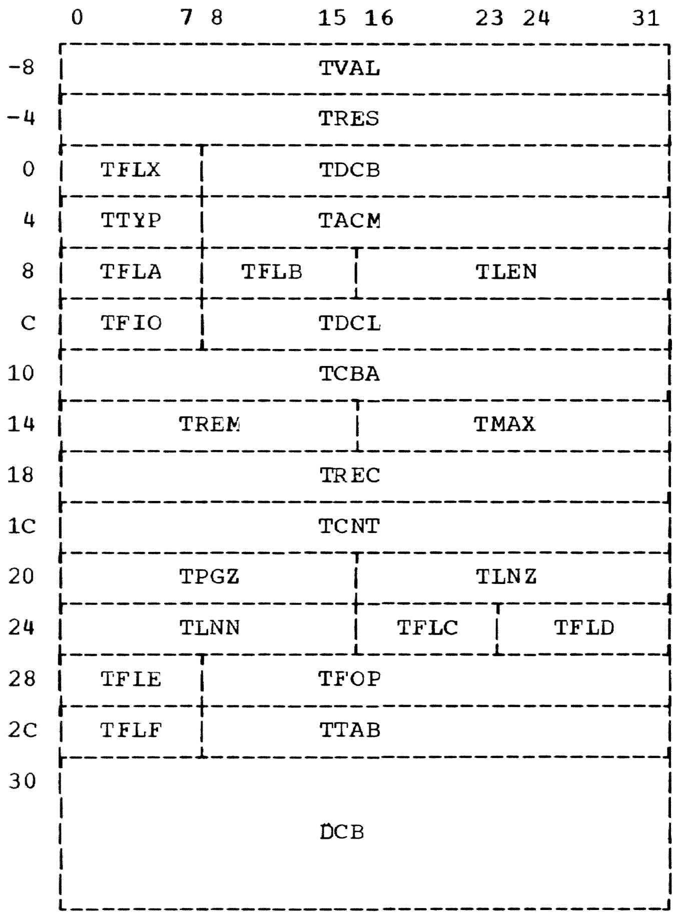


Figure 58. FCB for Stream-Oriented I/O

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.

TTYP
: eight-bit code specifying I/O type:

	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/TSWA			
18	TBBZ			
1C	TADC			
20	TLRR/TAID			
24	TLRL		TFLC	TFLD
28	TFLE	TFOP		
2C	TFLF	TFMP	(Reserved)	
30	TEVT			
34	Zero			
38	DCB			

Figure 59. FCB for Record-Oriented I/O

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 transit module, which interfaces with data management access methods. The names of all such library modules are IHEWIT*, 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
USASI 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
SYSOUT 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 USASI control byte (if present).

RECORD: TPKA
address of previous key. (Used for LOCATE creation of INDEXED data sets, and padding key for SEQUENTIAL INDEXED data sets.)

TSWA
address of data in dummy buffer got at OPEN time.

TREC/TBBZ

STREAM: TREC
address of buffer workspace (U-format output).

RECORD: TBBZ
length of IOCB.

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 DDEF command.

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 (in QSAM) this field holds the length of the previously read record if the previous operation was a READ SET.

TAID
address of dummy work area for terminal identification.

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 named TFDV and TFHE:

TFDV -- type of device

TFHE -- further file history

Meaning	Code TFDV	Code TFHE	Name
Paper tape	1000	0000	TMPA
Printer	0100	0000	TMPR
Conversational input flag	0010	0000	TMCI
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*
TELEPROCESSING	X'14'	TMTP*

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 PL/I program, or zero (if FCB is the first FCB opened).

TFLF
eight-bit code specifying the load module code:

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
QTAM	X'18'	TMQT
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. A TS loop is performed on this byte until it is freed. When the chaining operation is complete, the byte is set to zero.

----
*Not used in TSS/360.

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.

DCB
this field, variable in length and format, is the data control block defined by the data management for the various access methods.

INPUT/OUTPUT CONTROL BLOCK (IOCB)

This control block has three main divisions (see Figure 60). The size of the IOCB varies, as described in Figure 61, according to the access method. The fields in the IOCB are:

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.

BINO
chain-forward address of the next I/O control block.

BERR
flag byte for record-oreinted I/O situations:

Situation	Code	Name
IOCB has been checked	0000 0001	BMCH
I/O error exits	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 VISAM 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):

	0–7	8–15	16–31	
0	BACT	BPIO		IOCB foundation
4	BNIO			
8	BERR	BFCB		
C	BREQ			
	BERC/BEFC/BXTC/BKYC		BRCC	
14	BRVS			
18	BEVN			
1C	BDF1			
20	BDF2		BDF3	
24	BDF4			
28	BDF5			
2C	BECB			BSAM/VISAM DECB
30	BTYP		BLEN	
34	BDCB			
38	BARE			
3C	BSTS/BLOG			
40	BKVS/BKEY			
44	system use			
48	system use			
4C	system use			
50				
54				
.				BSAM hidden buffer area
.				
S1				

Note: (The IOCB includes the Data Event Control Block (DECB) for the BSAM and VISAM Interfaces)

Figure 60. Format of the I/O Control Block (IOCB)

	Sequential	Direct
	Consecutive	Indexed
F-format records	A B	A C D_1 D_2 16 (Note 1)
V-format records	A B D_2	-
U-format records	A B	-
A: Size of IOCB foundation B: Size of BSAM DECB C: Size of VISAM 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 61. Values Used in Computing Size of IOCB for Various Access Methods

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'20'	EVENT option
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 'Error and Interrupt Handling', in Section II, for details of these codes.)

BRCC
error code for RECORD condition.

(See 'Error and Interrupt Handling', in Section II, 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

BSAM: BDF1
address of the user's record variable.

BDF2

BSAM: BDF2
length, in bytes, of the user's record variable.

BDF3
BSAM
length, in bytes, of the KEYTO area.

BDF4

BSAM BDF4:
address of the KEYTO area.

BDF5: BSAM
relative record number (REGIONAL (1)).

DECB fields
see System Control Blocks PLM for detailed description.

BDBF (BSAM)
start of hidden buffer.

0	4	8	12
Type	0	Access	Mode

16	20	24	28 31
Flag A	Flag B	Flag C	Flag D

Figure 62. Format of the Open Control Block (OCB)

OPEN CONTROL BLOCK (OCB)

The format of the open control block (OCB) is shown in Figure 62. The fields in the OCB are:

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
Flag B		0	transient*
Flag C			(Reserved)
Flag D	Bit:	0	(Reserved)
		1	PRINT
		2	(Reserved)
		3	(Reserved)

----

*Not supported in TSS/360.

EXAMPLE OF CHAINING

Figure 63 contains an example of the chaining of FCBs, IOCBs, and event variables in a PL/I program.

Files

The program 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.

IOCBS

Two 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 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.

Event Variables

The program has three active I/O event variables. These are chained from the pseudo-register IHEQEVT so that, on termination of the program, 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 and 3 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 2, 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.

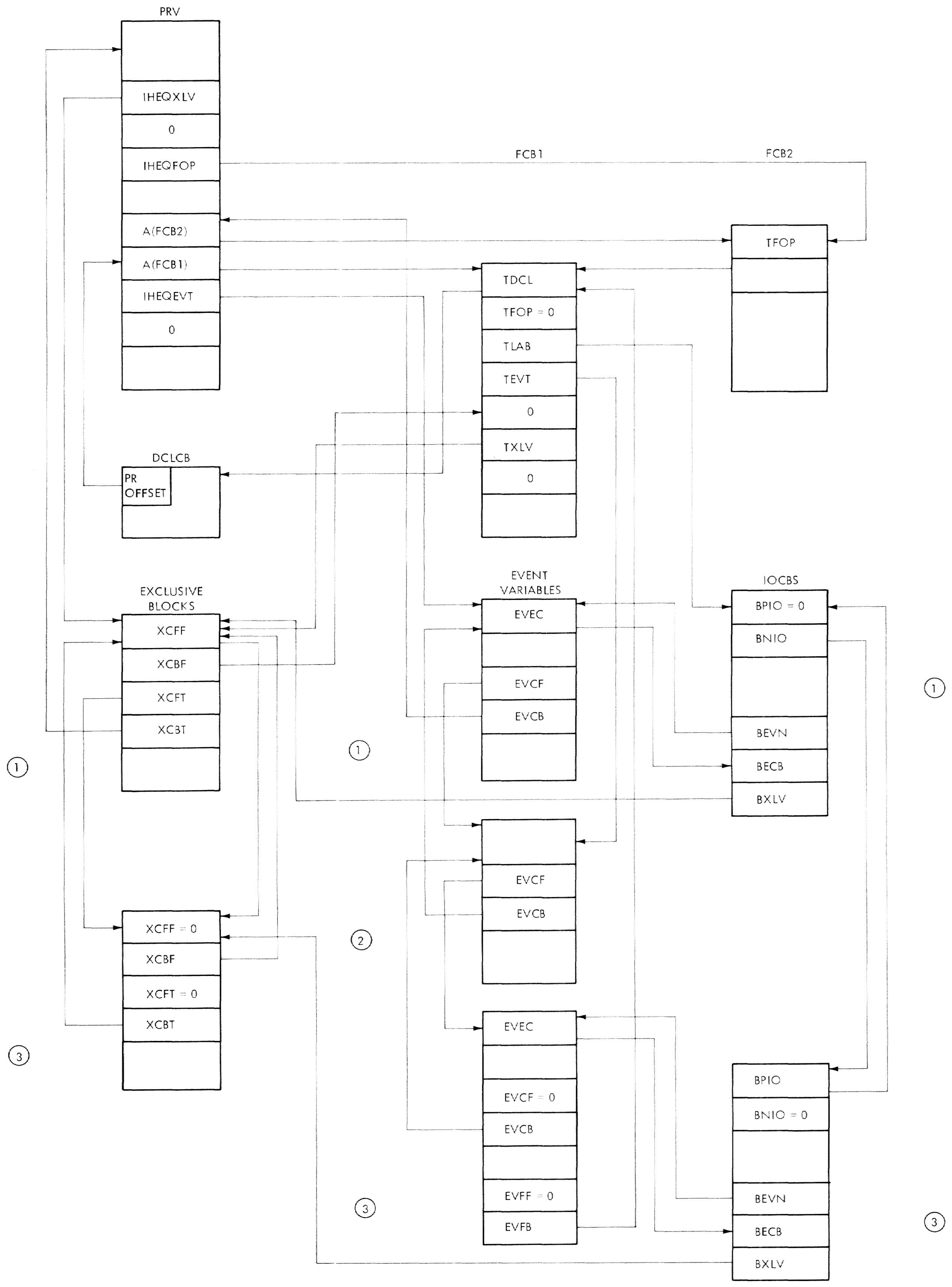


Figure 63. Example of Chaining of I/O Control Blocks

```
    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 64. Format of Area Variable

This describes the formats of the control blocks used by the storage-management modules of the PL/I Library. The functions of the blocks and the way they are used are described in Section II. In the diagrams, all offsets are in hexadecimal.

AREA VARIABLE

An area variable is prefaced by a four-word description of its extent (Figure 64).

DYNAMIC STORAGE AREA (DSA)

The format of a dynamic storage area is shown in Figure 65. The size of a DSA varies; the minimum size is X'64' bytes. The first byte contains flags (see Figure 66) describing the optional entries.

Standard Entries

Standard Save Area: The area starting with the flags and continuing up to and including the register save area. (See Figure 51 and associated text.)

Current File: This field is eight bytes long; its use is described in 'Current File' in 'Stream-Oriented I/O' in Section II.

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 earlier in this section, in 'Compiler-Generated Control Blocks'. The AUTOMATIC data, workspace and parameter lists areas are provided for use by the compiler.

VARIABLE DATA AREA (VDA)

A variable data area (see Figure 67) is a special type of automatic storage area, described in Section II. The first byte of the VDA contains flags (see Figure 68) describing the data. The PRV VDA is a VDA which contains the PRV and primary LWS. (See Figure 69). Secondary LWS is contained in the LWS VDA (Figure 70).

	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 65. Format of the Dynamic Storage Area (DSA)

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 66. Format of the DSA Flag Byte

	0 7	8 31
0	Flags	Length
4	Chain-back address	
8	Data	

Figure 67. Format of the Variable Data Area (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 68. Format of the VDA Flag Byte

	0 7	8 31
0	Flags	Length (= L(PRV) + L(LWS) + 8)
4	A (External save are)	
	Library workspace (LWS)	
	LSW (DSA optimization area, OPT=01 only	

Figure 69. Format of the PRV VDA

	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 70. Format of LWS VDA

SECTION IV

APPENDIXES

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	IHEWDUM, IHEWERR, IHEWZZC
CHECK	IHEWITB
CLOSE	IHEWCLT
DCB	IHEWOPO
DCBD	IHEWCLT, IHEWIOF, IHEWITB, IHE-WITD, IHEWITE, IHEWITG, IHEWITM, IHEWITN, IHEWOCL, IHEWOPO, IHE-WOPP, IHEWOPQ
DELREC	IHEWITD, IHEWITE, IHEWITM, IHEWITN
DIR	IHEWERR, IHEWSAP, IHEWZZC
EBCDTIME	IHEWCVC
ESETL	IHEWITD, IHEWITN
FINDJFCB	IHEWOPO
FREEMAIN	IHEWCLT, IHEWDSP, IHEWDUM, IHE-WITB, IHEWITG, IHEWLSP, IHEWOCL, IHEWOSW, IHEWSAP, IHEWZZC
FREEPOOL	IHEWCVC
GATRD	IHEWDSP
GATWR	IHEWCVC, IHEWDSP, IHEWSAP, IHEW-TOM, IHEWTSA, IHEWIOF
GET	IHEWIOF, IHEWITD, IHEWITG, IHEWITN
GETBUF	IHEWCVC
GETMAIN	IHEWDSP, IHEWDUM, IHEWITB, IHE-WITD, IHEWITE, IHEWITM, ITEWITN, IHEWLSP, IHEWOCL, IHEWOPO, IHE-WOPP, IHEWOPQ, IHEWSAP, IHEWZZC
GETPOOL	IHEWCVC
GTWRC	IHEWIOB, IHEWIOF, IHEWZZC
OBEY	IHEWZZC
OPEN	IHEWOPP
PAUSE	IHEWCVC
PUT	IHEWIOF, IHEWITD, IHEWITG, IHEWITN
PUTX	IHEWITG
READ	IHEWITE, IHEWITM
SETL	IHEWITD, IHEWITN
SIR	IHEWCVC, IHEWERR, IHEWSAP, IHEWZZC
SPEC	IHEWCVC, IHEWERR, IHEWSAP, IHEWZZC
STIMER	IHEWCVC
SYSIN	IHEWIOF
WRITE	IHEWITD, IHEWITE, IHEWITM, IHEWITN
XTRCT	IHEWIOB, IHEWIOF, IHEWIOP, IHEWZZC

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', Section II).

IHEQADC

Pointer to a list of address constants for use by the I/O routines: the list is in IHEWSAP.

IHEQATV

Not used in TSS/360.

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 'Stream-Oriented I/O' in Section II.

IHEQCTS

The base address of the non-sharable module IHEWCVC.

IHEQECA

Four byte interruption communication area.

IHEQERR

Serves as a parameter list when calling IHEERRB. The code associated with the ON condition to be raised is placed into IHE-QERR. See 'ON Conditions' in Section II, internal error codes Appendix D.

IHEQEVT

The anchor cell for the incomplete I/O event variables in a given PL/I program. When IHEQEVT contains zero, no I/O event variable in the PL/I program is incomplete.

IHEQFOP

The anchor cell of the chain linking the FCBs for the files opened in a given PL/I program. When IHEQFOP is zero, none of the files opened in this task are still open. See 'File Control Block' in Section IV.

IHEQFVD

Pointer to the Free VDA module in IHESAFD.

IHEQICA

Four byte interruption communication area.

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 Section IV.

IHEQLPR

Length of the pseudo-register vector. This is fixed, under TSS/360, and is 4096 bytes.

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 program information is saved (program mask, etc.). See Section IV.

IHEQLW0, IHEQLW1, IHEQLW2, IHEQLW3, IHEQLW4

Pointers to the various levels of library workspace; see 'Library Workspace' in Section IV.

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 'Execution-time Optimization' in 'Program Management' in Section IV.

IHEQRTC

Contains the return code used in the normal termination of a PL/I program.

IHEQSAR

Contains an environment count used by the display modification module (IHEWSAR) when on-units and entry parameter procedures are used in prologues and epilogues.

IHEQSFC

Pointer to free-storage within first block of storage obtained by the initialization library module (IHEWSAP).

IHEQSLA

Pointer to the storage area most recently allocated by the storage management routines. The area may be a DSA or a VDA.

IHEQSPR

The file register for SYSOUT, 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 Section II.

IHEQTIC

Not used in TSS/360.

IHEQVDA

Pointer to the Get VDA module: set (in IHEWSAP) to IHESADF.

IHEQXLV

Not used in TSS/360.

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.
Definitions of offsets in module IHEWCVC.

IHECVC

Operand: A four-character code denoting the last four letters of an entry point in the library.

Result: Register BR is loaded with the address of the entry point.

IHEYCVC

Operands: None

Result: Definition of all offsets in module IHEWCVC.

Used by: IHELIB and IHEWCVC.

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: LW0)
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.

IHEZAP

Operands: None

Result:

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 IHEWDUM, IHEWZZC, IHEWZZF.

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 IHEWERR in pseudo-register IHEQERR. This code is a 4-digit hexadecimal number. The two high-order digits denote the PL/I condition (Figure 71); the last digit may denote a specific error associated with that condition.

Code	Condition
1000	STRINGRANGE
1800	OVERFLOW
2000	SIZE
2800	FIXEDOVERFLOW
3000	SUBSCRIPTRANGE
3800	CHECK (label)
40XX	CONVERSION
4800	CHECK (variable)
5000	CONDITION (identifier)
5800	FINISH
6000	ERROR
6800	ZERODIVIDE
7000	UNDERFLOW
78XX	AREA
8800	NAME
90XX	RECORD
98XX	TRANSMIT
A000	I/O SIZE
A8XX	KEY
B000	ENDPAGE
B800	ENDFILE
C000	I/O CONVERSION
C8XX	UNDEFINEDFILE

Figure 71. 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 ON 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 E: DUMP INDEX

The dump index provided by the subroutines IHEWDUM, IHEWZZC and IHEWZZF contains information about:

Files currently open

Current file

Save areas

On-units, interrupts and other details

This information is output to SYSOUT.

If the task is conversational, the dump index is followed by a PAUSE. The programmer may enter any valid commands: for example, he may display areas defined by the index. Execution of the program continues after a GO command.

If the task is nonconversational, then all pages containing save areas or file blocks are dumped to SYSOUT, with DSNAME=PLIDUMP.

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:

```
 1  TDUMP:      PROC OPTIONS(MAIN);
 2              DCL A CHAR(4) INIT('ABCD');
 3              DCL IHESARC ENTRY(FIXED
                BIN);
 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(64);
17              Z=Y(32767,32767); /*
                ADDRESSING ERROR */
18              END TDUMP;
```

This program produces the following output and dump index when in conversational mode.

If there had been a current file, this would have appeared after the section on 'Files Opened by This Task.'

```
THIS IS THE FIRST LINE
THIS IS THE SECOND LINE
IHE8041 ADDRESSING INTERRUPT IN STATEMENT 00017 AT OFFSET +000AC FROM ENTRY POINT CONVPROC
                                 * * * TSS/360 PL/1 IHEDUMP   ID=    0 * * *
 *** FILES OPENED BY THIS TASK
 XYZ                            DCLCB 304000  FCB 350D30  DCB 350D68  PR OFFSET 0B0
 SYSPRINT                       DCLCB 303000  FCB 350C28  DCB 350C60  PR OFFSET 04C
 *** CHAIN BACK THROUGH SAVE AREAS
 3766F8  DSA FOR ERR  ON-UNIT                     CALLS IHEDUMP  FROM 3001C2 (STMT    5)
 376000  SECONDARY LIBRARY WORKSPACE
 376010  SAVE AREA FOR LIBRARY                    CALLS 300168   FROM 306428  LCA AT 3763F8
 363B58  SAVE AREA FOR LIBRARY                    CALLS 30504A   FROM 3063FA  LCA AT 363BF8
 363990  SAVE AREA FOR LIBRARY                           INTERRUPT AT 341056  LCA AT 363BF8
 301198  DSA FOR PROC CONVPROC                    CALLS 341000   FROM 3002CE (STMT   17)
 363EF8  DSA FOR ERR ON-UNIT RAISED BY CONV CONDITION. CALLS 300228   FROM 300222 (STMT    7)
 363800  SECONDARY LIBRARY WORKSPACE
 363810  SAVE AREA FOR LIBRARY                    CALLS 3001C8   FROM 306428  LCA AT 363BF8
 365350  SAVE AREA FOR LIBRARY                    CALLS 30504A   FROM 3063EA  LCA AT 3653F0
 3650F0  SAVE AREA FOR LIBRARY                    CALLS 306014   FROM 3430F2  LCA AT 3653F0
 365188  SAVE AREA FOR LIBRARY                    CALLS 343000   FROM 3410D6  LCA AT 3653F0
 301110  DSA FOR BEGIN                            CALLS 341000   FROM 300152 (STMT   13)
 3656F0  DSA FOR PROC TDUMP                       ENTERS BEGIN AT 300108
 364000  PRV - PSEUDO REGISTERS START AT 364008
 00247C  EXTERNAL SA                              CALLS 300000
```

APPENDIX F: PL/I LIBRARY MODULE NAMES AND ALIASES

This appendix contains a table listing the PL/I Library modules in alphabetical order along with their associated aliases; 6 character alias names are CSECTs, 7 character alias names are entry points. For a description of each module, see Section III, Module Summaries. In the interests of clarity, the preceding characters IHEW and IHE, as indicated by the first entries, have been omitted.

MODULE NAMES	ALIASES
IHEWABU	IHEABU0, IHEABU
ABV	ABV0, ABV
ABW	ABW0, ABW
ABZ	ABZ0, ABZ
ADD	ADD0, ADD
ADV	ADV0, ADV
APD	APDA, APDB, APD
ATL	ATL1,2,3 and 4, ATL
ATS	ATS1,2,3 and 4,ATS
ATW	ATWH, ATWN, ATW
ATZ	ATZH, ATZN, ATZ
BEG	BEGA, BEGN, BEG
BSA	BSA0, BSA
BSC	BSC0, BSC
BSD	BSD0, BSD
BSF	BSF0, BSF
BSI	BSI0, BSI
BSK	BSKA, BSKK, BSKR, BSK
BSM	BSMF, BSMV, BSMZ, BSM
BSN	BSNO, BSN
BSO	BSOO, BSO
BSS	BSS2 and 3, BSS
BST	BSTA, BST
BSV	BSVA, BSV
CFA	CFAA, CFA
CFB	CFBA, CFB
CFC	CFCA, CFC
CKP	CKPS, CKPT, CKP
CLT	CLTA, CLTB, CLT
CNT	CNTA, CNTB, CNT
CSC	CSC0, CSC
CSI	CSI0, CSI
CSK	CSKK, CSKR, CSK
CSM	CSMB, CSMF, CSMH, CSML, CSMV, CSM
CSS	CSS2 and 3, CSS
CST	CSTA, CST
CSV	CSVA, CSV
CVC	XCVC, CVC
DBN	DBNA, DBN
DCN	DCNA, DCNB, DCN
DDI	DDIA, DDIB, DDI
DDJ	DDJA, DDJ
DDO	DDOA, DDOB, DDOC, DDOD, DDOE, DDO
DDP	DDPA, DDPB, DDPC, DDPD, DDP
DIA	DIAA, DIAB, DIA
DIB	DIBA, DIBB, DIB
DID	DIDA, DID
DIE	DIEA, DIE
DIL	DILA, DILB, DIL
DIM	DIMA, DIM
DMA	DMAA, DMA
DNB	DNBA, DNB
DNC	DNCA, DNC
DOA	DOAA, DOAB, DOA
DOB	DOBA, DOBB, DOBC, DOB
DOD	DODA, DODB, DOD
DOE	DOEA, DOE
DOM	DOMA, DOM
DSP	DSPA, DSP
DUM	DUMC, DUMJ, DUMP, DUMT, DUM
DVU	DVU0, DVU
DVV	DVV0, DVV
DZW	DZW0, DZW
DZZ	DZZ0, DZZ
EFL	EFLC, EFLF, EFL
EFS	EFSC, EFSF, EFS
ERD	ERDA, ERD
ERE	EREA, ERE
ERI	ERIA, ERI
ERO	EROA, ERO
ERP	ERPA, ERP
ERR	ERRA, ERRB, ERRC, ERRD, ERRE, ERR
ESM	ESMA, ESMB, ESM
EXL	EXL0, EXL
EXS	EXS0, EXS
EXW	EXW0, EXW
EXZ	EXZ0, EXZ
HTL	HTL0, HTL
HTS	HTS0, HTS
IOA	IOAA, IOAB, IOAC, IOAD, IOAT, IOA
IOB	IOBA, IOBB, IOBC, IOBD, IOBE, IOBT, IOB
IOC	IOCA, IOCB, IOCC, IOCT, IOC
IOD	IODG, IODP, IODT, IOD
IOF	ITAZ, ITAX, IOFA, IOFB, ITAA, IOF
ION	IONA, ION
IOP	IOPA, IOPB, IOPC, IOP
IOX	IOXA, IOXB, IOXC, IOX
ITB	ITBA, ITB
ITD	ITDA, ITD
ITE	ITEA, ITE
ITG	ITGA, ITG
ITM	ITMA, ITM
ITN	ITNA, ITN
JXI	JXIA, JXII, JXIY, JXI
JXS	JXSI, JSXY
KCA	KCAA, KCA
KCB	KCBA, KCB
KCD	KCDA, KCDB, KCD
LDI	LDIA, LDIB, LDIC, LDI
LDO	LDOA, LDOB, LDOC, LDO
LNL	LNL2, LNLD, LNLE, LNL
LNS	LNS2, LNSD, LNSE, LNS
LNW	LNW0, LNW
LNZ	LNZ0, LNZ
LSP	LSPA, LSPB, LSPC, LSPD, LSPE, LSP
MPU	MPU0, MPU
MPV	MPV0, MPV
MXB	MXBN, MXBX, MXB
MXD	MXDN, MXDX, MXD

MXL	MXLN, MXLX, MXL
MXS	MXSN, MXSX, MXS
MZU	MZUD, MZUM, MZU
MZV	MZVD, MZVM, MZV
MZW	MZW0, MZW
MZZ	MZZ0, MZZ
NL1	NL1A, NL1L, NL1N, NL1
NL2	NL2A, NL2L, NL2N, NL2
OCL	OCLA, OCLB, OCLC, OCL
OPN	OPNA, OPN
OPO	OPOA, OPO
OPP	OPPA, OPP
OPQ	OPQA, OPQ
OSD	OSDA, OSD
OSE	OSEA, OSE
OSI	OSIA, OSI
OSS	OSSA, OSS
OST	OSTA, OST
OSW	OSWA, OSW
PDF	PDF0, PDF
PDL	PDL0, PDL
PDS	PDS0, PDS
PDW	PDW0, PDW
PDX	PDX0, PDX
PDZ	PDZ0, PDZ
PRT	PRTA, PRTB, PRT
PSF	PSF0, PSF
PSL	PSL0, PSL
PSS	PSS0, PSS
PSW	PSW0, PSW
PSX	PSX0, PSX
PSZ	PSZ0, PSZ
RES	REST, RESN, RES
SAP	SADA, SADB, SADD, SADE, SADF,
	SAFA, SAFB, SAFC, SAFD, SAFF,
	SAFQ, SAPA, SAPB, SAPC, SAPD,
	SARA, SARC, SAP
SHL	SHLS, SHLC, SHL
SHS	SHSC, SHSS, SHS
SMF	SMF0, SMF
SMG	SMGC, SMGR, SMG
SMH	SMHC, SMHR, SMH
SMX	SMX0, SMX
SNL	SNLC, SNLK, SNLS, SNL
SNS	SNSC, SNSK, SNSS, SNSZ, SNS
SQW	SQW0, SQW
SNW	SNWK, SNWC, SNWS, SNWZ, SNW
SPR	SPRT
SQL	SQL0, SQL
SQS	SQS0, SQS
SQZ	SQZ0, SQZ
SRC	SRCA, SRCB, SRCC, SRCD, SRCE,
	SRCF, SRC
SRD	SRDA, SRD
SRT	SRTA, SRTB, SRTC, SRT
SSF	SSF0, SSF
SSH	SSHC, SSHR, SSH
SSX	SSX0, SSX
STG	STGA, STGB, STG
STP	STPA, STP
STR	STRA, STRB, STRC, STR
TAB	TABS, TAB
TEA	TEAA, TEA
TEV	TEVA, TEV
THL	THL0, THL

THS	THS0, THS
TNL	TNLD, TNLR, TNL
TNS	TNSD, TNSR, TNS
TNW	TNWH, TNWN, TNW
TOM	TOMA, TOM
TSA	CTTA, DDTA, DDTB, DDTC, DDTD,
	DDTE, IBTA, IBTB, IBTC, IBTD,
	IBTE, IBTT, IGTA, INTA, OCTA,
	OCTB, OCTC, PTTA, PTTB, TCVA,
	TCVB, TERA, TPBA, TPRA, TSAA,
	TSAC, TSAD, TSAE, TSAF, TSAG,
	TSAL, TSAM, TSAN, TSAP, TSAR,
	TSAT, TSAV, TSAW, TSAX, TSAY,
	TSAZ, TSEA, TSSA, TSWA, TSA
UPA	UPAA, UPAB, UPA
UPB	UPBA, UPBB, UPB
VCA	VCAA, VCA
VCS	VCSA, VCSB, VCS
VFA	VFAA, VFA
VFB	VFBA, VFB
VFC	VFCA, VFC
VFD	VFDA, VFD
VFE	VFEA, VFE
VKB	VKBA, VKB
VKC	VKCA, VKA
VKF	VKFA, VKF
VKG	VKGA, VKG
VPA	VPAA, VPA
VPB	VPBA, VPB
VPC	VPCA, VPC
VPD	VPDA, VPD
VPE	VPEA, VPE
VPF	VPFA, VPF
VPG	VPGA, VPG
VPH	VPHA, VPH
VQA	VQAA, VQA
VQB	VQBA, VQB
VQC	VQCA, VQC
VSA	VSAA, VSA
VSB	VSBA, VSB
VSC	VSCA, VSC
VSD	VSDA, VSDB, VSD
VSE	VSEA, VSEB, VSE
VSF	VSFA, VSF
VTB	VTBA, VTB
XIB	XIB0, XIB
XID	XID0, XID
XIL	XIL0, XIL
XIS	XIS0, XIS
XIU	XIU0, XIU
XIV	XIV0, XIV
XIW	XIW0, XIW
XIZ	XIZ0, XIZ
XXL	XXL0, XXL
XXS	XXS0, XXS
XXW	XXW0, XXW
XXZ	XXZ0, XXZ
YGF	YGFS, YGFV, YGF
YGL	YGLS, YGL
YGS	YGSV, YGSS, YGS
YGW	YGWV, YGWS, YGW
YGX	YGXV, YGXS, YGX
YGZ	YGZV, YGZS, YGZ
ZZC	ZZCA, ZZC
ZZF	ZZFA, ZZF

APPENDIX G: PL/I SHARED LIBRARY ARRANGEMENT

The PL/I Library is arranged in two modules. Module CFBAI contains all the shared library modules, in control sections which are one page or less in length. Each control section is formed by linkage editing and is named by the first module linked into it. In so far as possible, modules in a control section refer only to each other and have no references outside the control section other than to module IHEWCVC, which contains all the address constants and is the only control section in module CFBAJ.

Module CFBAI (READONLY, PUBLIC, SYSTEM):

Control Section	Module Names															
ABW	ABW	SQS	SQW	ABZ	SQL	SQZ	ADD	ADV	APD	DVV	MPV	BSA	BSC	BSD	BSF	CFC
ATW	ATW	ATS	HTS	LNW	LNS	XXW	XXS	EXS	EFS	THS	TNW	TNS	SHS	EXW	SNW	SNS
ATZ	ATZ	ATL	HTL	LNZ	LNL	XXL	XXZ	EXZ	EXL	EFL	THL	TNZ	TNL			
BSK	BSK	STG	ERE	XIB	XID	XIL										
CST	CST	VSE	BSO	DNB	CSV	STP										
DDI	DDI	DDJ	DDO	DDP	MXL	MXS	OSE	OSS	YGL	YGS						
DIA	DIA	DIB	DID	DIE	DIM	DOA	DOB	DOD	DOE							
DMA	DMA	VFA	VFB	VFC	VFD	VFE	VKB	VKC	VPA	VTB	VPF	VPE				
DOM	DOM	DNC	DBN	DCN	KCA	KCD										
DZW	DZW	ZZF	ZZC	TSA	TOM											
ERR	ERR	BSS	CSS	DIL	SRC	SRD	YGW	CNT	BST							
ESM	ESM	CKP	ABV	BEG	XIZ	YGZ	YGX									
IOA	IOA	LDI	LDO	TEV	TEA											
IOX	IOX	IOB	IOC	IOD	IOF	ION	IOP									
ITB	ITB	SRT														
ITE	ITE	ITD														
ITN	ITN	CLT	OST													
JXI	JXI	NL2	PDL	PDS	PDW	PDZ	OSW	SMH	SMG	SMX	SMF	PDF	PDX	DVU	CSM	CSC
KCB	KCB	UPA	UPB	VCA	VCS	VSA	VSB	VSC	VSD							
OCL	OCL	OPN	OPQ	MZU												
OPO	OPO	MZV	MZW	MZZ	OSI	XIU	XIV	XIW	CFA							
OPP	OPP	XIS	ITM													
PRT	PRT	OSD	LSP	ITG	DUM	CFB										
SAP	SAP	DZZ														
SNL	SNL	SHL	SNZ	JXS	NLI	PSF	PSL	PSS	PSW	PSX						
SPR	SPR	PSZ	SSG	SSH	SSF	SSX	BSI	BSM	BSN							
VKF	VKF	VKG	VPB	VPC	VPD											
VPG	VPG	VPH	ERP	ERO	ERI	ABU										
VSF	VSF	VQA	VQB	VQC	ERD	CSI	CSK	MPV	MXB	MXD						
YGF	YGF	TAB	STR	DSP	BSV	RES										

Module CFBAJ (System)

Control Section: CVC

Modules: CVC

Where more than one page reference is given, the major reference is first.

GY28-2052-0

PL/I Subroutine Library Printed in U.S.A. GY28-2052-0

IBM

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]

IBM Technical Newsletter

File No.	S360-29
Re: Order No.	GY28-2052-0
This Newsletter No.	GN28-3192
Date:	September 30, 1971
Previous Newsletter No.	GN28-3162

IBM SYSTEM/360 TIME SHARING SYSTEM
PL/I SUBROUTINE LIBRARY
PROGRAM LOGIC MANUAL

This Technical Newsletter, a part of Version 8, Modification 1, of IBM System/360 Time Sharing System, provides replacement pages for the subject publication. Pages to be inserted and/or removed are as follows:

5,6	117,118
13-18	127,128
21,22	135-138

A change to the text is indicated by a vertical line to the left of the change.

Summary of Ammendments

Miscellaneous corrections are made.

Note: In this manual,

- Cross-references to Section II should be to Section I.
- Cross-references to Section III should be to Section II.
- Cross-references to Section IV should be to Section III.

Please file this cover letter at the back of the manual to provide a record of the changes.

IBM Corporation, Programming Publications, Dept. 643, Neighborhood Road, Kingston, N.Y. 12401

PRINTED IN U.S.A.