IBM Application Program

CALL/360-OS

PL/I System Manual - Volume IV

Program Number 360A-CX-45X

The CALL/360-OS PL/I compiler (to be used with the CALL/360-OS system on an IBM System/360 Model 50 or higher) is described in the four volumes of this publication. The publication is addressed to system programmers and customer engineers who require a detailed knowledge of the compiler. It contains a general overview of the compiler and detailed information on the compiler and runtime routines and macros that perform required functions. Additional information required to understand CALL/360-OS PL/I compiler operations is provided in several appendices. The appendices appear in this volume. They cover the following subjects:

Compiler conventions and data layout

Compiler tables and lists

Compiler support macros

Runtime support macros

Object code storage layout

Support services for language processors

CALL/360-OS PL/I compiler maintenance

Diagnostic messages

Maximum size of source program

Reference listings

Form No. GY20-0570-1

Terminal Equivalence

Terminals which are equivalent to those explicitly supported may also function satisfactorily. The customer is responsible for establishing equivalency. IBM assumes no responsibility for the impact that any changes to the IBM-supplied products or programs may have on such terminals.

Second Edition (January 1971)

This edition, GY20-0570-1, is a major revision obsoleting GY20-0570-1. It applies to Version 1, Modification Level 1, of CALL/360-OS and to all subsequent versions and modifications until otherwise indicated in new editions or Technical Newsletters.

Technical changes to text are indicated by vertical lines in the left margin. A revised illustration is indicated by the symbol • to the left of the caption. Page numbers in reference lists in Appendix J have been changed to reflect accurately the pages in the manual where the routines are discussed. Figure J-4, a module-module cross reference, has been added to this appendix.

Changes are continually made to the information herein. Therefore, before using this publication, consult the latest System/360 SRL Newsletter (GN20-0360) for the editions that are applicable and current.

Copies of this and other IBM publications can be obtained through IBM branch offices. A form has been provided at the back of this publication for readers' comments. If the form has been removed, address comments to: IBM Corporation, Technical Publications Department, 112 East Post Road, White Plains, New York 10601.

CONTENTS

FIGURES

APPENDIX A - COMPILER CONVENTIONS AND DATA LAYOUT

NAMING AND USAGE

The CALL/360-OS PL/I compiler is coded in OS/360 Level-F Assembler Language. To help clarify the relocation properties of the coding with respect to the special requirements of the CALL/360-OS operating environment, and to help make the organization of the compiler more apparent for maintenance purposes, certain symbolic naming conventions and usages have been observed throughout the coding as described below.

REGISTERS

The CALL/360-OS environment requires programs operating within it to be organized such that one group of registers can be relocated by the base address of the compiler and another group by the base address of the user's area, while a third group remains non-relocatable.

The nomenclature Gn has been used to indicate the non-relocatable (general) registers; Pn, the registers addressing the user's (program) area; and Cn, the registers addressing the compiler. General register zero is exempt from relocation by the system, and so remains a G-register, even though its physical assignment places it in the P-register group. The registers thus available to the compiler are:

G0, G2, G3, G4, G5, G6, and G7 (machine registers 0 and 2 - 7)
C1, C2, and C3 (registers 8 - 10)
P0, P1, P2, P3, P4, and P5 (registers 11 - 1, excluding 0)

In addition to distinguishing the registers by relocation property, certain compiler-wide register assignments are maintained.

C1 is used for subroutine linkages, both as entry point and return register.

C3 is used by most subroutines as the principal code cover register. A few large routines use C2 as a second cover register.

P2 permanently addresses the first 4096 bytes of the compiler's static working storage. This is the area containing all register save-areas, compiler-wide flags, switches, counters, etc., and adcons for all subroutines. P0 addresses the second 4096 bytes of this area.

P1 permanently addresses the base within the user's area at which object code will be generated. Since the contents of the compiler's working tables must be non-relocatable, wherever a true address would normally be used as a pointer, a non-relocatable displacement relative to the base contained in register P1 is used instead.

Registers P0, P1, and P2 are never used for any other purpose by any subroutine. Across subroutine calls, the called routine is responsible for savinG all G- and C-registers except G0, and all P-registers except P5. The floating-point registers are assigned mnemonics of F1, F2, F3, and F4; they are non-relocatable and, in view of the rarity of their use, are considered volatile across subroutine calls.

SUBROUTINES

Subroutine entry points are named $xxxxx, where xxxxx is a mnemonically suggestive symbol. The adcon which addresses the entry point is named @xxxxx. Each subroutine has been assigned a two-letter prefix for use in creating local labels. Thus, the END Generator, for example, uses a prefix of ED, the Instruction Assembler, VN, and so on. Labels on instructions within the subroutine are constructed according to the format ppnnn, where pp is the routine's two-letter prefix, and nnn is a sequence number assigned as closely as possible in ascending order throughout the routine.

Local working storage (used only by the routine in question) is identified with symbols of the form ppxxxx, where pp again is the routine's two-letter prefix, and xxxx is a mnemonic.

Note: The CALL/360-OS PL/I compile-time subroutine entry point names follow the naming conventions stated above and applied in this manual. However, there are some exceptions in the member names assigned to certain routines when stored in CALL/360-OS PL/I system libraries. For the reader's convenience, the exceptions are noted in Figure J-4, which is a cross reference of compilation module calls to other compilation modules.

REGISTER SAVE-AREAS

Each subroutine must have three register save-areas. Each class of registers must be saved in storage which has the matching relocation attribute. Register save-areas are named:

W$Cxx
W$Pxx
W$Gxx

where the C, P, and G indicate the relocation class, and xx is the two-letter prefix used by the subroutine which saves the registers.

COMPILER-WIDE VARIABLES

Communication between subroutines sometimes involves the use of flags, switches, counters, and other discrete variables which are independent of the main data tables used by the compiler. These variables are located in the compiler's fixed-size working storage area. This area is permanently addressed by registers P0 and P2 and is subdivided by relocation property into three parts: a C-area, a P-area, and a G-area, corresponding to the relocation properties of the general registers. Within these three areas, symbolic names are used mnemonically, with first characters of $ for ordinary variables and @ for adcons. (Register save-area names begin with W$; see above.) Except for the register save-areas, each compiler-wide variable is individually described under "Compiler Variables."

COMPILER TABLES AND LISTS

The principal data used during compilation are kept in tables and lists. (See Appendix B.) To facilitate the naming of all pertinent fields in these tables, the following symbolic conventions are used.

Most tables and lists are assigned single-letter prefixes, p.

Each field within such a table or list is named p$xxxx where xxxx is from one to seven characters mnemonically suggestive of the field's use.

Absolute values associated with a table or field are named p@xxxx.

Logical masks used to extract data from a field are p#xxxx.

Thus, for example, the dictionary attribute list has a prefix of A; the data definition information field in this list is named A$DEF; the mask used to extract type information is named A#DEFS; and the value code used to identify a contextual declaration is named A@CTXT.

SYMBOLIC ORGANIZATION

The subroutines which comprise the compiler are individually assembled. They are link-edited together to form two phases. The first phase contains all routines necessary to support compilation and code-generation. The second phase contains the compiler routines necessary to complete the initialization and initiation of the object code, together with a control copy of all object-program library support routines. The object-time library routines needed by a given compilation are effectively "loaded" by the second phase of the compiler. Communication of table structure and working storage layouts for the compiler is achieved through use of the Symbol Definition macro (SYMDEF), which is one of a set of assembler-language macros written especially to support the CALL/360-OS PL/I compiler (see Appendix C).

RUNTIME ROUTINE STRUCTURE

Because of the nature of the Runtime Library Loader routine ($HRTLL), all CALL/360-OS PL/I library runtime routines are structured according to certain conventions. The basic layout of a routine is illustrated below.

```
          1       2       3       4        Byte
        ┌──────────────────────────────────┐
Word 1  |     LENGTH     |   EXT  |   ENT   |
        |----------------------------------|
        |          C(LENGTH) bytes          |
        |          of machine-language      |
        |          code                     |
        |                                   |
        .                                   .
        .                                   .
        .                                   .
        |                                   |
        |----------------------------------|
        |          External Reference       |
        |          or Jump Table            |
        |                                   |
        |          Length is C(EXT)         |
        |          halfwords.               |
        .                                   .
        .                                   .
        .                                   .
        |                                   |
        |----------------------------------|
        |          Entry Point Table        |
        |                                   |
        |          Length is C(ENT)         |
        |          fullwords.               |
        .                                   .
        .                                   .
        .                                   .
        └──────────────────────────────────┘
```

Values for Fields:

LENGTH — Size of the module, excluding the first word and the two trailing tables.

EXT — Number of halfword entries in the external reference (jump) table.

ENT — Number of fullword entries in the entry point table.

External Reference or Jump Table — One entry is made to this table for each unique external reference in the library runtime routine. The entry contains the library load number of the referenced routine. If more than one entry point of a routine is referenced, a unique entry is made for each entry point. (See "Library Search ($NLSIB)" in Volume I for more information about library load numbers.)

Entry Point Table — Each word of the entry point table has the following format:

```
             1       2       3       4      Byte
         ┌───────────────────────────────────┐
  Word 1 │       NUMB      │       DISP      │
         └───────────────────────────────────┘
```

NUMB - Library load number for the entry point.

DISP - Displacement of that entry point from the beginning of the object code for this library runtime routine.

Values for these fields are usually generated by means of the Header macro (IHEHDR) and Trailer macro (IHETLR). The external reference and entry point tables are referenced when a library runtime routine is loaded, but they are not actually loaded with the routine.

COMPILER VARIABLES

All compiler-wide variables other than register save-areas are described in this subsection. These variables are located in the C-area, P-area, and G-area of the compiler's fixed-size working storage.

C-AREA

Variables in the C-area are relocated using the base address of the compiler. They are as follows:

M$ — Address of the symbolic instruction table (in module $TCODE).

O$ — Address of the operation code table (in module $TCODE).

$XSAVE — Save-area for registers C1 and C2, used as required by the compiler support macros and the expandable-table support subroutines ($WBACK, $WSTEP, $WCTCT, $WEXP).

$BASE — Address of first byte of current phase of compiler.

P-AREA

Variables in the P-area are relocated using the base address of the user's (program) area. They are as follows:

$COMAD — Contains address of communications area.

$PSCRT — Pair of scratch words, used mainly by entokening phase in forming offset within user area.

$SCNX — Scan-index, contains address of next character in source program following last semicolon entokened.

$TSA — Address of first word boundary within source program. Used as starting address of dope vector list for compilation wrap-up.

W$PNS2 — Used by $NCONS as save-area for register P5.

G-AREA

Variables in the G-area are not relocatable. They are as follows:

$ACODE — Pointer to next available byte in object code area. High byte contains object code base identification (@ACODE).

$ASC — Offset to next available byte in static and constants area. High byte contains static and constants base identification (@ASC). Initialized to allow beginning of static and constants area to be free for use as DSA of external procedure.

$ASCA — Offset to next available byte in static array and string storage. High byte contains the base code for this area (@ASCA).

$AADCN — Offset to next available byte in adcon storage. High byte contains the base code for this area (@AADCN). Initialized past preallocated part of adcon storage.

$DISPL — Contains displacement from variable tables address (register P1) to fixed tables address (register P2). Used in creating pointers to items in fixed tables area.

$CSS — Compound Statement Switch - indicates whether a unit of a compound statement needs to be completed immediately. This switch is checked just before generating triads for each statement. Switch has four settings:

@CSSOF	No units to complete.
@CSSON	Must complete an on-unit.
@CSSTN	Must complete a THEN-unit.
@CSSES	Must complete an ELSE-unit.

$NIDSI — Identifier Search Indicator - used by Locate Variable routine ($FVAR) to determine type of identifier desired. Settings:

= 0	Variable
= 4	Filename
= 8	Label constant or variable
= 12	Entry name
= 255	Return from $FVAR if file created.

$CHRFG Building Character String Switch - used by Increment Scan Index routine ($ASIDX) to determine whether source line being crossed is in middle of a character string. Settings:

= 0	Not in middle of string.
≠ 0	In middle of string.

$CLBLS Label Switch - indicates whether a statement label needs to be processed. Settings:

= 0	No label on statement
= 1	Statement label
= 2	Begin label
= 3	Entry label
= 4	Format label

$EOS End of Source Switch - used by entokening phase to determine whether all of the source program has previously been used. Settings:

= 0	Not all used.
≠ 0	All used.

$CCF Compilation Completed Flag - used by Increment Scan Index routine ($ASIDX) to determine whether to generate new line tokens when crossing line boundaries. Settings:

= 0	Compilation not completed; build tokens.
≠ 0	Compilation completed; do not build tokens.

$TAREA Translate Area - used by entokening phase to contain translate and test tables.

$LLINE Last Line - used by Controller ($CNT) and entokening routines. Contains new line token for last source line for which a line number table entry was made.

$CLPTR Label Pointer - if $CLBLS ≠ 0, $CLPTR contains pointer to statement label token.

$ABTBL Attribute Table - used in declaration processing to indicate attributes which the Attribute Node Creation routine ($ANCRE) should use in creating an attribute entry.

Each attribute table consists of an attribute bit string and pointers for the various attributes. If an attribute was specified without its list, a corresponding bit is set in the attribute bit string, but the pointer for the attribute is zero.

The format of the attribute table is shown below.

```
                1         2         3         4     Byte
           r--------------------------------1
           |                                |
    Word 1 |   Attribute Bit String         |
           |                                |
           |--------------------------------|
           |                                |
         2 |   Pointer to Environment List  |
           |                                |
           |--------------------------------|
           |                                |
         3 |   Pointer to Returns List      |
           |                                |
           |--------------------------------|
           |                                |
         4 | Pointer to String Length List  |
           |                                |
           |--------------------------------|
           |                                |
         5 |   Pointer to Entry List        |
           |                                |
           |--------------------------------|
           |                                |
         6 |   Pointer to Precision List    |
           |                                |
           |--------------------------------|
           |                                |
         7 |  Pointer to Dimension List     |
           |                                |
           L--------------------------------J
```

$APARM Identifier a Parameter - used in declaration processing to indicate whether Attribute Node Creation routine ($ANCRE) should allocate storage for identifier (see also $APRMA). Settings:

= 0	Not parameter; allocate storage.
≠ 0	Parameter; do not allocate storage.

$APRMA Parameter Address - used in declaration processing to contain the address of a parameter being declared (see also $APARM).

$BIOTY I/O Type - used in I/O processing to indicate the type of I/O being compiled. Settings:

Output	Edit	= 4
	List	= 8
	Data	= 12
Input	Edit	= 16
	List	= 20
	Data	= not set

$CBKNO Current Block Number - contains the identification number of the block currently being compiled. The identification number for the external procedure block is 32 (X'20'), and other blocks are numbered ascendingly as encountered.

$CBKCT Current Block Count - contains the count of the number of blocks encountered in the source program plus 31. Used to assign identification numbers to new blocks.

$DCNME Declaration Name - during declaration processing, contains a pointer to name entry for the identifier being declared.

$DSKIP Skip Pointer - during I/O processing, contains pointer to the SKIP token, if present; otherwise zero.

$DFILE File Pointer - during I/O processing, contains pointer to the FILE token, if present; otherwise zero (location is $DSKIP+4).

$DDATA Data List Pointer - during I/O processing, contains pointer to the LIST, DATA, or EDIT token, if present; otherwise zero (location is $DFILE+4).

$DOBY DO BY Clause - during loop processing, contains the type (second byte), precision (third byte), and scale (fourth byte) of BY clause expression. Second word contains result of expression processor evaluation of BY clause (unless constant provided, in which case a constant token is present).

$DOLHS DO Left Hand Side - during loop processing, contains, in same format as $DOBY, indication of iteration variable for loop.

$DORHS DO Right Hand Side - during loop processing, contains, in same format as $DOBY, indication of initial setting for loop interation variable.

$DOTO DO TO Clause - during loop processing, contains, in same format as $DOBY, indication of TO value for loop.

$EXPCT Expansion Count - during array expression expansions, contains the number of DO-loops generated for the expansion.

$DOSWT DO Switch - indicates which of the TO and BY clauses are present for the loop. Setting:

Bit 0:	= 0	No TO clause ($DOTO not set.)
	= 1	TO clause ($DOTO set.)
Bit 1:	= 0	No BY clause ($DOBY not set.)
	= 1	BY clause ($DOBY set.)

$FEDC FED a Constant - indicates if a format element descriptor (FED) being considered is all constant. Setting:

= 0	All constant.
≠ 0	Not all constant.

$FED FED - during Format Item routine ($FORI), contains skeletal FED. Upon exit from $FORI, contains address of FED.

$FEDNM FED Number - contains number of expressions in FED currently being processed.

$FCB FCB - during GET and PUT processing, contains pointer to attribute entry for the file.

$FORAD Format Address - during format processing, contains the address of a pair of words in the adcon area used by the format.

$PTR Pointer to Token Table - used to communicate a token table pointer between routines.

$APARAM Previous parameter identification.

$DIO Not used.

$DDC Not used.

$DDO Not used.

$DSP Not used.

$DBS Not used.

$ERROR Parameter list for Error Message Editor ($XERR); first word contains a pointer to error token, next three words are optional pointers to token or N list entries, and last two words contain a character string literal parameter.

$FCBAD Fixed adcon address of FCB common area in a form for code generation (0C000010).

@FBKNO Value for first block number.

$GABK Blank character. Must immediately precede $GABU. Used to clear $GABU print area.

$GABU Print buffer for most messages from compiler to terminal.

$HECVD Not used.

L$LIBX Displacement from the start of the adcon area to the first available adcon following the fixed adcon area.

$LIBBC Base code for library, base code = @LIBBC.

$LTEND Length, in bytes, of the library load table.

@LOAD Displacement from start of fixed tables to library load table.

$MFCB Mask for FCB control bytes.

$MLWS Displacement from code to address modifiable LWS.

MTIO Value of displacement from communications area to terminal I/O buffer.

$PARAM Address of parameter table shared by library routines referenced by a fixed adcon.

@PSIZE Number of words in initially allocated library parameter table.

$PSIZE Number of words in current library parameter table.

$TDUMP Not used.

$LNTA Displacement from code to line number table.

$CAA Displacement from code to static and constants area.

$AAA Displacement from code to adcon area.

$RTLA Displacement from code to library area.

$LWSA Displacement from code to LWS area.

$SASA Displacement from code to static array and string storage area.

$I/OBA Displacement from code to disk I/O buffer area.

$DSAA Displacement from code to DSA area.

EP1 Number of bytes that must be between end of code and variable tables at end of the entokening of a statement in order not to cause request for more space.

EP2 Number of bytes that must be between end of code and variable tables at beginning of code generation for a statement in order not to cause request for more space.

$VIN Not used.

$K1 to $K8 Not used.

$CTON Count of current nesting of on-units.

$EXPNS Information passed from Expression Processor Controller ($NEXP) to Expander routine (see $EXPND). This is a two-word entry. The first word contains the comma count and the second, the attribute pointer.

$LASTL Last line number for which a line number table (D table) entry was made.

$TSOFF Pair of words used by the entokening phase to calculate offset for new line tokens and offsets within line.

$NPVF Complex Pseudo-Variable Flag - used by Expression Processor Controller ($NEXP) during an assign to a complex pseudo-variable. If the value is 12, the left side of the assignment symbol is the complex pseudo-variable.

$OBJLC Equated to $ACODE, the location counter associated with the code.

$NLINE New Line Flag. If 1, a call to a library routine has previously been encountered in the current source line.

$NC1W Head of the chain of constant table entries whose length is 4 bytes.

$NC2W Head of the chain of constant table entries whose length is 8 bytes.

$NC4W Head of the chain of constant table entries whose length is 16 bytes.

$NCMSC Head of the chain of constant table entries whose length is not 4, 8, or 16 bytes.

$NCPXP Used by Expression Processor Controller as save-area for arguments and right side of complex pseudo-variable.

$NDIG Used by Constant Conversion routine ($NCVT) to form a floating-point value of a digit of a source constant. The word is preset with an exponent of two and the value zero.

$PTO Origin of the table of operands which are parameters to the symbolic instruction passed by Triad Code Generator ($TCODE) to the Instruction Assembler ($VINSA). There are six entries in this data parameter table, each with the following format:

```
             1        2        3        4        Byte
           ┌───────────────────────────────────┐
    Word 1 │              $PTKN                │
           │───────────────────────────────────│
         2 │              $PADD                │
           │───────────────────────────────────│
         3 │$PSGN  │$PREG   │      $PLNG       │
           └───────────────────────────────────┘
```

$PTKN The operand token. The first byte is the type byte and the other three bytes contain a quantity which is dependent upon the type. These operand tokens are identical to the operands of a triad.

$PADD Core address of the operand. The first byte indicates the relocation base and the other three bytes the offset to the operand within the data area.

$PSGN Contains five flags:

X '01' #PSGN If 1, the operand is positive in core.

X '10' If 1, the operand is address of FCIB for TITLE move.

X '20' #PRS If 1, the operand is negative in a register.

X '40' #PRAM If 1, the operand is a parameter of a subprogram.

X '80' If 1, the operand requires a dummy argument.

$PREG Register address of the operand. Contains the pointer to the register table entry which contains the operand.

$PLNG If the operand is a string, this field contains the length in bytes.

$NROPN Operand area of the Expression Processor Controller which contains the attributes and description of the right operand. The format is:

```
                1            2       3        4         Byte
           ┌─────────────────────────────────────────────┐
           |            |         |       $NROL        |
    Word 1 |            | $NROTM  |- - - - - - - - - - |
           |            |         | $NROPR  | $NROSF   |
           |---------------------------------------------|
         2 |                    $NRTKN                   |
           |---------------------------------------------|
         3 | $NROPI   |          $NRPTR                  |
           └─────────────────────────────────────────────┘
```

$NROTM Right Operand Type Mask. The bits have the same meaning as the dictionary attribute data description field.

$NROL Right operand length if it is a string.

$NROPR Right operand precision if its type is arithmetic.

$NROSF Right operand scale factor if its type is arithmetic.

$NRTKN, $NROPI, and $NRPTR These fields are set from the operand stack and have the same meaning.

$NLOPN Operand area of the Expression Processor Controller which contains the attributes and description of the left operand. The format is the same as for the right operand. The fields are $NLOTM, $NLOL, $NLOPR, $NLOSF, $NLTKN, $NLOPI, and $NLPTR.

$NRSLT Operand area of the Expression Processor Controller which contains the attributes and description of the result of an operation. The format is the same as for the right operand. The fields are $NRTM, $NRL, $NRPR, $NRSF, $NTKN, $NRPI, and $NPTR.

$NO Maps an area with the same format as the right operand area. The fields are $NOTM, $NOL, $NOPR, $NOSF, $NOTKN, $NOPI, and $NOPTR.

$NXOP An operand area with the same format as the right operand area. Used in building triads which do not come directly from the operator and operand tokens of the stacks. The second and third words of this area are referenced by the labels $NXTKN and $NXPTR.

$NYOP An operand area with the same format as the right operand area. Used in building triads which do not come directly from the operator and operand tokens of the stacks. The second and third words of this area are referenced by the labels $NYTKN and $NYPTR; the type mask is referenced using the label $NYTM.

$PRIOR Contains the priority of the operator pending addition to the operator stack.

$NAARG Contains the count of the number of arguments in an argument list which are arrays. Used in processing calls to the array built-in function POLY.

$NTCUR The number of the last triad generated.

$NCCUR The number of the last triad for which code has been generated.

$NBIF Built-In Function Flag. If 0, function not built-in; if 1, built-in function and convert all arguments to the result type; if 2, built-in function and arguments require special conversion.

$NFLAG Fixed-Point Scale Flag. If 1, scale value to result scale converting to result type.

$NXFLG Communication Flag - used by several routines for various purposes.

$NEXPT Indicates type of expression result required by a statement processor. Has the same meaning as dictionary attribute description field except that if the byte is all ones, any type is satisfactory.

$TCD Used to dump the fixed table area on entrance to Triad Code Generator ($TCODE).

$TCA Used to dump the fixed table area on each triad processed by $TCODE.

$EXA Used to dump the fixed table area on each token processed by $NEXP.

W$GTC2 Save-area for the data parameter ($PTO) table in $TCODE when dope vectors must be generated for string arguments of built-in functions.

$HEADS A table of pointers to the beginning-of-segment control word of the first segment of each expandable table maintained by the compiler. The individual table pointers are as follows:

Node Code

04 - - - C$HEAD	Constant table
16 - - - D$HEAD	Line number table
14 - - - I$HEAD	Initialization table
06 - - - B$HEAD	Block information table
0E - - - E$HEAD	Error message table
13 - - - L$HEAD	Library load table
07 - - - P$HEAD	Program structure table
18 - - - Q$HEAD	Subscript substitution table
09 - - - S$HEAD	Temporary storage table
08 - - - T$HEAD	Token table
19 - - - V$HEAD	Expression stack
01 - - - X$HEAD	Operator stack
02 - - - Y$HEAD	Operand stack
03 - - - Z$HEAD	Triad table

A$HEAD Pointer to the beginning of the dictionary attribute list (A list).

H$HEAD Pointer to the beginning of the dictionary hash table (H table).

N$HEAD Pointer to the beginning of the dictionary name list (N list).

J$HEAD Pointer to the beginning of the supplementary initialization list (J list).

$TAILS A table of pointers to the end-of-segment control word of the last currently active segment of each expandable table maintained by the compiler. The individual table pointers have names of the format P$TAIL, and match the sequence given above for $HEADS.

A$TAIL Pointer to the last dictionary attribute node.

H$TAIL Pointer to the last dictionary hash table node (not used).

N$TAIL Pointer to the last dictionary name list node.

J$TAIL Pointer to the last node in the supplementary initialization list.

$ACTVS A table of pointers to the currently available data space position within each expandable table segment maintained by the compiler. The individual pointers have names of the format P$ACTV and match the sequence given above for $HEADS.

$CURRS A table of two-word pointers associated with each expandable table in the compiler. These pointers support the non-destructive GPREV macro and the GNEXT macro. (See Appendix C.) The first word of the pair points to the beginning-of-segment control word for the given segment of the table currently being scanned. The second word points to the end of the data space within the given segment of the table currently being scanned. Together, the pointers serve as limits in each direction for the segment scanning macros. The individual pointers have names of the form P$CURR and match the sequence given above for $HEADS.

SEGLST A pointer to the beginning-of-segment control word of the first free expandable-table segment. If no segments are free, SEGLST contains zero.

FREPTR A pointer to the first available (unused) word in the compiler's variable data space. Data space is always acquired by decrementing FREPTR, since working storage grows from higher-numbered storage locations toward lower-numbered ones.

R$TBL Pointer to the base of the register table (R table).

R$FX Pointer to the fixed-point-register portion of the register table.

R$FL Pointer to the floating-point-register portion of the register table.

R$AD Pointer to the adcon register portion of the register table.

R$ND Pointer to the end of the linear portion of the register table.

R$SY Pointer to the head of the register table synonym list. This list contains unassigned synonym entries. Assigned synonyms are detached from this list and attached to the appropriate register table entry. Twenty synonym entries are available. If all are in use, R$SY contains zero.

$TEMPL Temporary storage level count. Initially zero, it is increased by one at the beginning of a DO-loop and is decreased by one at the end.

R$ARRC Reference count for adcon register assignment. Initially zero, it is increased by one each time an adcon register is assigned. It effects a rotational assignment of the available adcon registers.

$VSART Table of symbolic adcon register assignments used by the Instruction Assembler routine ($VINSA).

$VSCRT Table of symbolic computational register assignments used by $VINSA.

$VLBLT Table of local symbolic labels used by generated instruction sequences used by $VINSA.

$VRAMT Flag for register assignment: 0 indicates single register; nonzero, double. Used in calling the Computational Register Assignment routine ($VASGC).

$VRTYP Flag for register type assignment: a high-order bit of 0 indicates a floating-point register; 1, a fixed-point register. Used in calling $VASGC.

$VLPAK Doubleword-aligned work area. Used principally by Error Message Editor ($XERR) for unpacking and conversion operations.

$VLINE Print-line work area. Used by $XERR to format output lines.

$SEVCO Highest severity code encountered by $XERR during processing of error messages. Initially zero.

$SEVCT Total number of error messages produced during compilation. Initially zero.

$VLS Not used.

$TEMPN Level number to be assigned to temporary storage associated with the saving of registers around a DO-loop. $TEMPN is set by the "begin DO" pseudo-operation in Instruction Assembler ($VINSA) and used by the Temporary Storage Management routine ($VGTMP).

$TITLE Dummy title attribute entry (6 words long).

$FILEON On-unit flag and FCIB pointer.

$NESTK Address of top of expression stack.

$DBUF Number of disk buffers needed.

$LOAD Library load table.

APPENDIX B - COMPILER TABLES AND LISTS

GENERAL

With a few exceptions, fixed-size tables are either located within the fixed area of working storage or assembled as part of a compiler module. Some of the tables within the fixed area of working storage are discussed under "Compiler Variables" in Appendix A. Tables within a compiler module are unmodifiable as well as fixed. The symbol table that is within the Triad Code Generator routine ($TCODE) is discussed in Appendix E.

Items within the variable portion of working storage, with a few exceptions, are either expandable tables or lists. Lists (by definition) have a variable number of entries. For a discussion of expandable tables and lists, see "Table Handling Macros" in Appendix C. Unless otherwise specified, the tables discussed below are located in the variable portion of working storage.

TITLE: DICTIONARY ATTRIBUTE LIST (A LIST)

Purpose and Usage

The dictionary attribute entry is created for each definition of an identifier in the source program. The attribute entry contains all of the information needed by the compiler about the identifier.

Usage Description

Each dictionary name entry points to a list (possibly null) of attribute entries. Contained in this attribute list are all definitions of the identifier made in blocks still in the process of translation. The attribute entries are stored as a list, with the last entry pointing to the name entry. If there is more than one attribute for an identifier, the attribute entries are ordered inversely by block number.

Attribute entries are also created to describe the parameter requirements for entry names. Each entry name attribute entry points to a list of attribute entries specifying parameter requirements. An attribute entry for an entry name also contains a pointer to the RETURNS attributes used when the entry name is referenced.

Each block information table (B table) entry for a procedure block points to the RETURNS attributes for the block as defined in the PROCEDURE statement.

Entry Description - General

There are five general types of attribute entries. Thirteen bytes of each entry are standardized. Figure B-1 shows the first 13 bytes of every attribute entry other than a constant or built-in function name entry. (See succeeding discussion for details.)

```
                1           2           3             4         Byte
          ┌─────────────────────────────────────────────────┐
          | Node Type |  Pointer to Next Attribute           |
   Word 1 | A$NODE    |           A$NEXT                     |
          |   =A@     |                                      |
          |-----------+--------------------------------------|
          |Identifier| Block    | Last Block | Definition    |
        2 |   Type   | Declared |    Used    | Information   |
          |  A$TYPE  |  A$BDCL  |   A$LBLK   |   A$DEF       |
          |-----------+--------------------------------------|
          |  Address  |                                      |
        3 | Base Code |          Address Offset              |
          |  A$BASE   |             A$DISP                   |
          |-----------+--------------------------------------|
          |   Data    |                                      |
        4 | Descriptor|                                      |
          |   A$DD    |                                      |
          |-----------+                                      |
          |                                                  |
          |                                                  |
          |                                                  |
           - - - - - - - - - - - - - - - - - - - - - - - - -
```

Figure B-1. Dictionary Attribute Entry--First 13 Bytes

The first four bytes of this area contain the standard node type and pointer to next attribute entry. The A$TYPE byte contains a code indicating the type of identifier. The values of this code are:

A@STRG	4	Denotes character-string variable.
A@REAL	6	Denotes a real variable.
A@CPLX	7	Denotes a complex variable.
A@LBLV	8	Denotes a label variable.
A@LBLC	9	Denotes a statement-label constant.
A@ENTR	10	Denotes an entry name.
A@FILE	11	Denotes a filename.

The block number of the block in which the identifier was declared is contained in A$BDCL. The A$LBLK byte contains the block number of the last block in which the identifier was used.

Definition information is contained in A$DEF. The values assumed on the bases of logical tests are given below.

Mask and Bit	Value	=0	=1
A#PARM	0	not parameter	parameter
A#SCOP	1	internal	title move external scope
A#TEMP	2	not temporary	temporary storage
A#USED	3	not used	used
A#SET	4 5	not set static	set automatic
A#DEFS	6 7	=00 tentative (A#TENT) =10 implicit (A#IMPL)	=11 explicit (A#EXPL) =01 contextual (A#CTXT)

A$BASE and A$DISP contain the address of the identifier. The first byte of this word (A$BASE) contains a code describing the base address to be used, and the last three bytes (A$DISP) contain the offset from this base. The code numbers and the corresponding bases are as follows:

Code No.	Base
4	Object
8	Static
A	Array
C	Adcon

The A$DD byte contains a data descriptor. This descriptor is defined separately for each attribute entry type.

Nonlabel Variable Entry

Entries for nonlabel variables are either five or six words in length depending on whether the variable is dimensioned. A$REG contains a register number indicating whether the identifier is in a register and, if so, which one. A$REGS contains the sign. A$DIMS contains the number of dimensions. A$LNG has the length of a string, if known. The precision and scale of arithmetic variables are in A$PREC and A$SCAL.

If the identifier is dimensioned, a sixth word contains dimension bound codes in A$DC and a pointer to the dope vector as contained in the compiler's static storage initialization list in A$DVP. Bit 0 of the dimension code is 0 if all bounds of the array are constants and 1 if not. Bits 1 to 7 indicate whether both bounds for the first to seventh dimensions, respectively, are constants. If the ith bit is zero, then both bounds are constant for the ith dimension (1 ≤ i ≤ 7).

If the nonlabel variable is a parameter, the address field contains the address of an adcon where the address of the variable or its dope vector is stored. If the variable is not a parameter, then it contains the address of the variable or its dope vector directly.

If the attribute entry is for an array, then A$DD contains a special descriptor X'08' and the array element descriptor is placed in A$DDE.

Figure B-2 shows the layout of an attribute entry for a nonlabel variable.

```
                1          2           3          4          Byte
          ┌───────────┬─────────────────────────────────────┐
          │  A$NODE   │       Pointer to Next Attribute     │
   Word 1 │   =A@     │              A$NEXT                 │
          │───────────┼─────────────────────────────────────│
          │Identifier │   Block     │ Last Block │Definition│
        2 │   Type    │  Declared   │    Used    │Information│
          │  A$TYPE   │   A$BDCL    │   A$LBLK   │  A$DEF   │
          │───────────┼─────────────────────────────────────│
          │  Address  │          Address Offset             │
        3 │ Base Code │             A$DISP                  │
          │  A$BASE   │                                     │
          │───────────┼─────────────────────────────────────│
          │   Data    │             │  Register  │ Number of│
        4 │ Descriptor│  Register   │    Sign    │Dimensions│
          │   A$DD    │   A$REG     │   A$REGS   │  A$DIMS  │
          │───────────┼─────────────┼───────────────────────│
          │           │   Array     │        Length         │
          │           │  Element    │        A$LNG          │
        5 │ Not Used  │ Descriptor  │───────────────────────│
          │           │   A$DDE     │   A$PREC   │ A$SCAL   │
          │           │             │  Precision │ Scale    │
          │───────────┼─────────────────────────────────────│
          │ Dimension │       Pointer to Dope Vector        │
        6 │   Codes   │             A$DVP                   │
          │   A$DC    │                                     │
          └─────────────────────────────────────────────────┘
```

Figure B-2. Dictionary Attribute Entry for Nonlabel Variable

Data Descriptor for Nonlabel Variable:

```
    If string = '01000100' B.
    If arithmetic:
       bit 0 = 1 (Arithmetic)A#ARTH
           1 = 1 (Variable)A#VRBL
           2 = 0 (Non-error)A#ERR
           3 = 0 Short⎫
             = 1 Long ⎭ A#LONG
           4 = 0 (Ordinary)A#SPCL
           5 = 0 (Non-String Type)A#STRG
           6 = 0 Fixed⎫
             = 1 Float⎭  A#FLT
           7 = 0 Real   ⎫
             = 1 Complex⎭  A#CPLX
```

Label Variable Entry

Each label variable entry is either four or six words long depending on whether it is dimensioned. The entry's structure is similar to that of a nonlabel variable. A$DIMS contains the number of dimensions. If the variable is an array, the sixth word contains dimension bound codes and a pointer to the dope vector. Both of these are the same as described for nonlabel variables.

If the label variable is a parameter, the address field contains the address of an adcon where the address of the variable or its dope vector is stored. If the variable is not a parameter, it contains the address of the variable or its dope vector directly.

Figure B-3 shows the layout of an attribute entry for a label variable.

```
               1          2          3            4        Byte
         r-------------------------------------------------1
         |Node Type |   Pointer to Next Attribute          |
  Word 1 | A$NODE   |            A$NEXT                    |
         |   =A@    |                                      |
         |----------+--------------------------------------|
         |Identifier|  Block    |Last Block |Definition    |
       2 |  Type    | Declared  |   Used    |Information   |
         | A$TYPE   |  A$BDCL   |  A$LBLK   |   A$DEF      |
         |----------+--------------------------------------|
         | Address  |                                      |
       3 | Base Code|          Address Offset              |
         | A$BASE   |            A$DISP                    |
         |----------+--------------------------------------|
         |   Data   | Register  | Register  | Number of    |
       4 |Descriptor|  A$REG    |   Sign    |Dimensions    |
         |   A$DD   |           |  A$REGS   |  A$DIMS      |
         |----------+-----------+--------------------------|
         |          |  Array    |                          |
       5 | Not Used | Element   |        Not Used          |
         |          |  A$DDE    |                          |
         |----------+--------------------------------------|
         |Dimension |                                      |
       6 |  Codes   |     Pointer to Dope Vector           |
         |  A$DC    |            A$DVP                     |
         L-------------------------------------------------J
```

Figure B-3. Dictionary Attribute Entry for Label Variable

Data Descriptor for Label Variable = '00001000' B.

Statement-Label Constant Entry

Each statement-label constant entry is four words long. In addition to the standard information, it contains a one-byte field A$LC containing label codes. These codes and the layout of the entry are shown in Figure B-4.

```
              1            2           3           4          Byte
         ┌----------------------------------------------------┐
         | Node Type |      Pointer to Next Attribute         |
  Word 1 |  A$NODE   |              A$NEXT                    |
         |   =A@     |                                        |
         |-----------+----------------------------------------|
         | Identifier|   Block    |Last Block |Definition     |
       2 |  Type     | Declared   |  Used     |Information    |
         |  A$TYPE   |  A$BDCL    | A$LBLK    |  A$DEF        |
         |-----------+----------------------------------------|
         |  Address  |                                        |
       3 | Base Code |          Address Offset                |
         |  A$BASE   |             A$DISP                     |
         |-----------+----------------------------------------|
         |   Data    | Register   | Register  |  Label        |
       4 | Descriptor|  A$REG     |   Sign    |  Codes        |
         |   A$DD    |            |  A$REGS   |  A$LC         |
         └----------------------------------------------------┘
```

Figure B-4. Dictionary Attribute Entry for Statement-Label Constant

Label Codes:

Bit 0 & 1:	= 00	Statement label A#STMT
	= 01	Begin label A#BEG
	= 11	Format label A#FRMT

Data Descriptor for Statement-Label Constant = '00001001' B.

The address word contains the address of the statement if the definition is explicit or the address of last usage if the definition is tentative.

Entry Name Entry

Each attribute entry for an entry name is five words long. A$RETP contains a pointer to an attribute node that contains the RETURNS attributes of the entry name. These RETURNS attributes are those used when the entry name is referenced and not those used inside the procedure on the occurrence of a RETURN statement. (These attributes are in the block information table.)

A#PRMS contains the number of parameters for the entry name and A$PRMP contains a pointer to the attribute list for the parameters. This list contains one entry for each parameter. If the data descriptor in one of these attribute entries is zero, the attributes for the associated parameter are unspecified. If the attributes for the parameters are not specified, the list is null.

If the entry name is itself a parameter, the address contains the address of an adcon where the address of the block adcon area (BAA) is stored. If the entry name is not a parameter, it contains the address of the BAA.

The format for an entry name attribute node is given in Figure B-5.

<table>
<tr><th></th><th>1</th><th>2</th><th>3</th><th>4</th><th>Byte</th></tr>
<tr><td>Word 1</td><td>Node Type
A$NODE
=A@</td><td colspan="3">Pointer to Next Attribute
A$NEXT</td><td></td></tr>
<tr><td>2</td><td>Identifier Type
A$TYPE</td><td>Block Declared
A$BDCL</td><td>Last Block Used
A$LBLK</td><td>Definition Information
A$DEF</td><td></td></tr>
<tr><td>3</td><td>Address Base Code
A$BASE</td><td colspan="3">Address Offset
A$DISP</td><td></td></tr>
<tr><td>4</td><td>Data Descriptor
A$DD</td><td colspan="3">Pointer to RETURNS Attribute Code
A$RETP</td><td></td></tr>
<tr><td>5</td><td>Number of Parameters
A$PRMS</td><td colspan="3">Pointer to Parameters List
A$PRMP</td><td></td></tr>
</table>

Figure B-5. Dictionary Attribute Entry for Entry Name

Data Descriptor for Entry Name = '00001011' B.

If the entry name is a built-in function, the address base code is zero and the rest of word 3 and words 4 and 5 contain information describing the function.

Built-In Function Entry Name Entry

Each attribute entry for a built-in function is five words long. The format of each entry is shown in Figure B-6.

<table>
<tr><th></th><th>1</th><th>2</th><th>3</th><th>4</th><th>Byte</th></tr>
<tr><td>Word 1</td><td>Node Type</td><td colspan="3">Pointer to Next Attribute</td><td></td></tr>
<tr><td>2</td><td>Identifier Type</td><td>Block Declared</td><td>Last Block Used</td><td>Definition Information</td><td></td></tr>
<tr><td>3</td><td>0</td><td>Choice Type</td><td colspan="2">Built-In Number</td><td></td></tr>
<tr><td>4</td><td>Data Descriptor</td><td>Result Type</td><td>Result Size</td><td>In-Line Number</td><td></td></tr>
<tr><td>5</td><td>Number of Parameters</td><td>Function Type</td><td>Argument Conversion Type</td><td>Flags</td><td></td></tr>
</table>

Figure B-6. Dictionary Attribute Entry for Built-In Function Entry Name

The first two words are identical in meaning to those of a non-built-in function entry name entry. Other entries are initialized to the values given below.

	Entry Name	BICT Choice Type	BIN Built-In Number	BIRT Result Type	BIRS Result Size	BINO In-Line Number	PRMS Number Parameters	BIFT Function Type	BIAC Argument Convert (Hex)	BIFC Flags (Hex)
FLOAT										
	EXP	0	EXS0	0	0		1	0	02	24
	LOG	0	LNS0	0	0		1	0	02	24
	LOG10	0	LGS0	0	0		1	0	02	A4
	LOG2	0	L2S0	0	0		1	0	02	A4
	ATAN	12	ATS0	0	0		1	0	02	34
	TAN	0	TNS0	0	0		1	0	02	24
	SIN	0	SNS0	0	0		1	0	02	24
	COS	0	CSS0	0	0		1	0	02	24
	TANH	0	THS0	0	0		1	0	02	24
	ERF	0	EFS0	0	0		1	0	02	A4
	SQRT	0	SQS0	0	0		1	0	02	24
	COSH	0	CHS0	0	0		1	0	02	24
	SINH	0	SHS0	0	0		1	0	02	24
	ATANH	0	AHS0	0	0		1	0	02	24
STRING										
	CHAR	8		16	20	4	1	1	FF	30
	SUBSTR	8	C'SR'	16	20	72	2	1	FF	38
ARRAY										
GENERIC	SUM	0	SMS0	20	0		1	2	FF	00
	PROD	0	PDS0	20	0		1	2	FF	00
	POLY	0	YGSS	20	0		2	2	00	00
	LBOUND	8		4	0	8	2	2	FF	00
	HBOUND	8		4	0	12	2	2	FF	00
	DIM	8		4	0	16	2	2	FF	00
MISCEL-										
LANEOUS	DATE	8	C'DE'	16	24	80	0	1	FF	00
	TIME	8	C'TE'	16	28	84	0	1	FF	00
ARITHMETIC										
GENERIC	ABS	4	ABT0	0	4	20	1	0	00	20
	MAX	0	MXS0	0	8		255	0	00	A0
	MIN	0	MNS0	0	8		255	0	00	A0
	MOD	8		0	12	24	2	0	00	A0
	SIGN	8		4	0	28	1	0	00	A0
	FLOOR	8		0	16	32	1	0	00	A0
	CEIL	8		0	16	36	1	0	00	A0
	TRUNC	8		0	16	40	1	0	00	A0
	COMPLEX	8		12	8	48	2	0	FE	28
	REAL	8		8	0	56	1	0	01	28
	IMAG	8		8	0	64	1	0	01	28
	CONJG	8		12	0	44	1	0	01	20

Choice Type (A$BICT) indicates how to choose the specific routine entry point for a call. This field has the following meanings and values:

0 No in-line expansions; choose routine by highest argument type.
4 In-line expansions for real arguments; choose complex routine by highest argument type.
8 All argument types expanded in-line.
12 Choose entry point by number of arguments and argument type if arithmetic.

Built-In Number (A$BIN) is the library basic entry point number of the function. All numbers of a built-in function are ordered so that it is possible to compute the number associated with all entry points of the function from the value of this field. An entry point number is selected on the basis of the attribute(s) of the argument(s). It is used to create an entry in the library load table. The real, float, single precision entry point is given if the routine is arithmetic.

Note: Entries for A$BIN are expressed as four-character names. This appears to conflict with the "Built-In Number" classification.

In the Phase 1 Initializer ($CCONT), a group of four characters is used as the last part of a symbol which is equated to the built-in number. The first two characters of the symbol are L@. Thus, for EXP (the first built-in function listed above), the code to generate its built-in number is:

```
DC  AL2(L@EXS0)
```

Result Type (A$BIRT) indicates the means of determining the attributes of the returns value. This field has the following values and meanings:

0 Result is the highest type of the arguments.

4 Result is a fixed integer.

8 Result is real and highest argument type.

12 Result is complex and highest argument type.

16 Result is a character string.

20 Result is float and highest argument type.

Result Size (A$BIRS) indicates the means of determining the precision or length of the returns values. This field has the following values and meanings:

0 Length or precision is determined by argument.

4 Fixed complex precision is (MIN(9,p+1),q).

8 Fixed precision is (MIN(9,MAX(all p-q))+MAX(all q)),MAX(all q).

12 Fixed precision is (MIN(9,r-s+MAX(q,s)),MAX(q,s))

16 Fixed precision is (MIN(9,MAX(p-q+1,1)),0).

20 SUBSTR or CHAR length.

24 Length is 6.

28 Length is 9.

In-Line Number (A$BINO) is the number identifying the built-in function. The in-line number of the pseudo-variable of the same name is obtained by adding four to this number. This field has the following values and meanings:

4	CHAR	(52)*	COMPLEX pseudo-variable
8	LBOUND	56	REAL
12	HBOUND	(60)*	REAL pseudo-variable
16	DIM	64	IMAG
20	ABS	(68)*	IMAG pseudo-variable
24	MOD	72	SUBSTR
28	SIGN	(76)*	SUBSTR pseudo-variable
32	FLOOR	80	DATE
36	CEIL	84	TIME
40	TRUNC	(88)*	Arithmetic to string conversion
44	CONJG	(92)*	String to arithmetic conversion
48	COMPLEX		

*Do not appear in an attribute entry but appear as triad operands.

Number of Parameters (A$PRMS) contains the number of parameters required for a function reference. If an optional additional parameter is possible, the count is for the minimum number of arguments. If the number of arguments is variable (for example, MAX and MIN), this field is set to its maximum value.

Function Type (A$BIFT) indicates the general classification of the function. The field has the following values and meanings:

0 Arithmetic

1 String

2 Array

<u>Argument Conversion Type</u> (A$BIAC) indicates the conversion required for the arguments. The field has the following values and meanings:

X'00'	Convert to highest argument type.
X'02'	Convert to highest argument type and float.
X'01'	Convert to highest argument type and convert to complex.
X'FE'	Convert to highest argument type and convert to real.
X'FF'	Do not convert arguments.

<u>Flags Field</u> (A$BIFG) contains a series of one-bit flags with the following meanings and settings:

	<u>Bit</u>	
A#CMPX	0	If 1, complex arguments are not allowed.
	1	If 1, add scaling information as an argument.
A#ARG	2	If 1, array argument causes array assign.
A#XARG	3	If 1, function may have optional extra argument.
A#PSEU	4	If 1, name may be a pseudo-variable.
A#AOK	5	If 1, function name may be an argument.

<u>Filename Entry</u>

Each filename attribute entry is four words long. A$FC contains a file code that specifies the attributes of the file.

If the filename is a parameter, the address field contains an address of an adcon that contains the address of the file control interface block (FCIB) for the file. If the filename is not a parameter, then the third word contains the address of the FCIB in the static and constants area.

Figure B-7 shows the format of an attribute entry for a filename.

	1	2	3	4	Byte
Word 1	Node Type A$NODE =A@	Pointer to Next Attribute A$NEXT			
2	Identifier Type A$TYPE	Block Declared A$BDCL	Last Block Used A$LBLK	Definition Information A$DEF	
3	Address Base Code A$BASE	Address Offset A$DISP			
4	Data Descriptor A$DD	Not Used	File Codes A$FC	Not Used	

Figure B-7. Dictionary Attribute Entry for Filename

File Codes:

```
bits 0 & 1 = 00 - SYSIN
             01 - DISK INPUT
             10 - SYSPRINT
             11 - DISK OUTPUT

     bit 2 =      DISK INPUT
         3 =      DISK OUTPUT
         4 =      DISK ENVIRONMENT
         5 =      NOT USED
         6 =  0 - EXTERNAL
              1 - INTERNAL
         7 =  0 - NON-PRINT
              1 - PRINT
```

Data Descriptor for Filename = '00001100' B.

Constant Attribute Entry

For each constant appearing in a source statement, a constant attribute entry is created. This attribute entry contains all of the attributes of the constant that can be implied from its EBCDIC form. The constant attribute entries look like normal attribute entries in order to simplify later processing. Constant attribute entries are stored in the token table areas so their space can easily be released at the end of usage.

Each constant attribute entry is six words long. The first five words are exactly the same as for a normal attribute entry except that A$BDCL and A$LBLK do not contain the block declared and used information. If the converted constant has different attributes from the source attributes (as in the case of compile-time conversions), the attributes entries (ADD, ALNG, A$PREC, and A$SCAL) are modified accordingly. However, a copy of the original attributes is retained in what is normally the address word.

Since a constant is always undimensioned, the sixth word is used to contain a pointer to the beginning of the source representation of the constant and its length in bytes, if arithmetic. If the constant is a string, the length in the source is not given since it can be longer than 255 characters (for example, if the string contains embedded quotes, each of which must be represented by two single quotes).

Figure B-8 shows the format of a constant attribute entry.

```
                  1           2           3           4       Byte
           ┌-------------------------------------------------┐
           | Node Type |                                     |
    Word 1 |  A$NODE   |              Not Used               |
           |   =AC@    |                                     |
           |-----------+-------------------------------------|
           | Identifier|                         |Definition |
         2 |   Type    |        Not Used         |Information|
           |  A$TYPE   |                         |  A$DEF    |
           |-----------+-------------------------------------|
           |           |           |    Original A$LNG       |
           |    Not    | Original  | - - - - - - - - - - - - |
         3 |   Used    |   A$DD    |  Original   | Original  |
           |           |           |  A$PREC     |  A$SCAL   |
           |-----------+-----------+-------------+-----------|
           |   Data    |           |             |           |
         4 | Descriptor| Register 1|     Not     |   Zero    |
           |   A$DD    |   A$REG   |    Used     |           |
           |-----------------------+-------------------------|
           |                       |         Length          |
         5 |       Not Used        |- - - - - - - - - - - - -|
           |                       | Precision   | Scale     |
           |-------------------------------------------------|
           |  Length   |                                     |
         6 | of Source |          Pointer to Source          |
           |  A$SRCL   |              A$SRCP                 |
           └-------------------------------------------------┘
```

Figure B-8. Dictionary Attribute Entry for a Constant

Data Descriptor: Same as for nonlabel variables, except bit 1=0.

TITLE: BLOCK INFORMATION TABLE (B TABLE)

Purpose and Usage

The block information table contains one entry for each unterminated block. This table contains all block-related information.

Description

The block information table is maintained as an expandable table. Each entry is eight words long.

The table contains one entry for each block still in the process of compilation (including the external procedure). The table is treated like a push-down list.

Entry Format

The layout of a block information table entry is shown below.

	1	2	3	4	Byte
Word 1	Node Type B$NODE	Pointer to RETURNS Attributes B$RETP			
2	Block Number B$BLNR	Symbol Table Switch B$STSW		Number of On-ENDFILEs Received B$FILE	
3	Address of Adcon Covering Symbol Table B$STA				
4	Address of BAA B$BAA				
5	Block Number	Amount of DSA Used B$DSAM			
6	Pointer to Constant Table Entry for DSA-Size B$DSAS				
7	Address of Last Prologue Link B$LPLA				
8	Address of Prologue Termination B$PTA				
9	Address of On-Unit Parameter List B$ON				

Values for Fields:

```
B$STSW   Symbol Table Switch (Test with B#STSW.)
               Bit 0: = 0 No symbol table needed.
                      = 1 Symbol table needed.
```

B$NODE For begin block, BB@ = 8.
For procedure block, BP@ = 4.

The first word of each entry contains a node type and a pointer to the RETURNS attribute node for the block if it is a procedure. These RETURNS attributes are those declared in the PROCEDURE statement. If the block is a begin block, the last three bytes of the first word are null.

The second word contains the block number of the block and a switch determining whether to produce a symbol table for the block at the end of the block. A count is also kept of the number of ON ENDFILEs encountered within the block. If a symbol table is to be produced, the third word contains the address of an adcon that is to contain the address of the symbol table.

The fourth word contains the address of the BAA for the block.

In the fifth word is a count of the number of bytes in the DSA that are assigned. This count is from the beginning of the DSA and includes all of the bookkeeping bytes. The top byte of this word contains a DSA address base code which is the block number.

The sixth word contains a pointer to a constant table entry that is to be initialized to the size of the DSA. This initialization is performed at the end of the block after the size of the DSA is known. (This size does not include any space for arrays or strings. Space for these is obtained separately.)

Words seven and eight are concerned with the chain of prologue instructions running through the block. The seventh word contains the address of the last branch in the chain. This branch still needs to be resolved. The eighth word contains the address of the end of prologue instructions. These instructions immediately precede the first executable statement. At the end of the block the chain is closed by resolving the last branch with the end of prologue address.

The ninth word contains an address for the on-unit parameter list. Three words (two of which are subsequently used for the list) are obtained when first encountering an ON ENDFILE statement within the block. Additional groups of three words are obtained for any other ON ENDFILE statements in the block. (See "On-Unit Parameter List" in Appendix B.)

TITLE: CONSTANT TABLE (C TABLE)

Purpose and Usage

The constant table contains an entry for each constant required in the object program. The arithmetic and all alphameric constants less than 16 bytes which do not contain primes, or are split between lines, are entered in their converted binary representation. The alphameric constants not converted are entered as pointers to their appearance in the source code.

Description

The constant table is maintained as an expandable table. The entries of the table are linked together according to length into four lists for searching; 4 bytes, 8 bytes, 16 bytes, and all others. The heads of these lists are pointed to from $NC1W, $NC2W, $NC4W, and $NCMSC, respectively.

Entry Format

The format of an entry containing a converted constant is shown below. (The value of C$LNK does not equal 1.)

```
                1        2        3        4           Byte
            ┌─────────────────────────────────────┐
     Word 1 │  C$REG   │          C$ADDR           │
            │──────────┼───────────────────────────│
          2 │  C$CNT   │        C$LNK              │
            │──────────┼───────────────────────────│
          3 │                   C$VAL1             │
            │──────────────────────────────────────│

          4 │                   C$VAL2             │

            │- - - - - - - - - - - - - - - - - - - │

          5 │                   C$VAL3             │

            │- - - - - - - - - - - - - - - - - - - │

          6 │                   C$VAL4             │

            └- - - - - - - - - - - - - - - - - - - ┘
```

Values for Fields:

C$REG — If nonzero, the register table entry which contains the value of the constant.

C$ADDR — The offset in static storage to the value of the constant.

C$CNT — The number of bytes in the constant.

C$LNK — Pointer to the next constant table entry in its search list if nonzero. If zero, the entry is the last element of the list.

C$VAL1 — First four bytes of constant value.

C$VAL2 — Second four bytes of constant value if the value of C$CNT is greater than 4.

C$VAL3 Third four bytes of constant value if the value of C$CNT is greater than 8.

C$VAL4 Last four bytes of constant value if the value of C$CNT is greater than 16.

The format of an entry containing a pointer to an unconverted constant is shown below. (The value of C$LNK equals 1.)

```
               1          2          3          4      Byte
         ┌──────────────────────────────────────────┐
  Word 1 │   C$REG    │            C$ADDR           │
         ├────────────┼─────────────────────────────┤
       2 │   C$CNT    │            C$LNK            │
         ├──────────────────────────────────────────┤
       3 │                 C$SPTR                   │
         └──────────────────────────────────────────┘
```

Values for Fields:

C$REG Same as described above.

C$ADDR Same as described above.

C$CNT Same as described above.

C$LNK Always equal to one.

C$SPTR Pointer to the first byte of the constant in the source program.

TITLE: LINE NUMBER TABLE (D TABLE)

Purpose and Usage

The line number table is the input list to the Line Number Table Processor ($HLNTP) and is used to generate the line number table, which relates the instruction addresses to the source line numbers.

Description

This expandable table contains one entry for each line in the source program. Entries are ordered by line number and machine address.

Entry Format

Each entry contains the following fields:

1. Pointer to the first character (in the source program area) of the line number.

2, Object code address.

These entry fields are shown below:

```
                1       2       3       4       Byte
           ┌─────────────────────────────────┐
           |        |       |       |        |
    Word 1 | Pointer to First Character of   |
           |     Line Number (D$LNP)         |
           |        |       |       |        |
           |---------------------------------|
           |        |       |       |        |
         2 | Object Code Address (D$OCA)     |
           |        |       |       |        |
           └─────────────────────────────────┘
```

TITLE: DICTIONARY HASH TABLE (H TABLE)

Purpose and Usage

The dictionary hash table is the directory to the dictionary name lists and is a fixed-length contiguous-entry table.

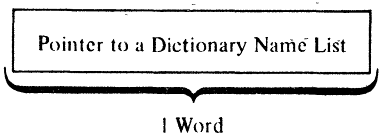
Note: In contrast to most tables in the variable area, this table is not expandable.

Description

Each entry in the dictionary hash table is a fullword pointer to a dictionary name list containing entries for all names that hash to the same value. A hash table entry is located by assuming that the first four characters of the name are an integer value and dividing this value by the number of entries in the hash table. The remainder thus obtained is then an index to the hash table entry.

Entry Format

The format of each entry in the dictionary hash table is shown below.



TITLE: INITIALIZATION TABLE (I TABLE)

Purpose and Usage

The initialization table is generated by various routines in the compiler as the program is being compiled. This table gives the Static Constants-Adcon Loader routine ($HSCAL) the information for initializing the static-constant and adcon areas.

Description

This is an expandable table having seven types of entries. The first byte (I$NODE) is used to distinguish the entry type.

Entry Format

1. Immediate value node

	1	2	3	4	Byte
Word 1	I$NODE	I$ASAD			
2	I$DATA				
. . .					

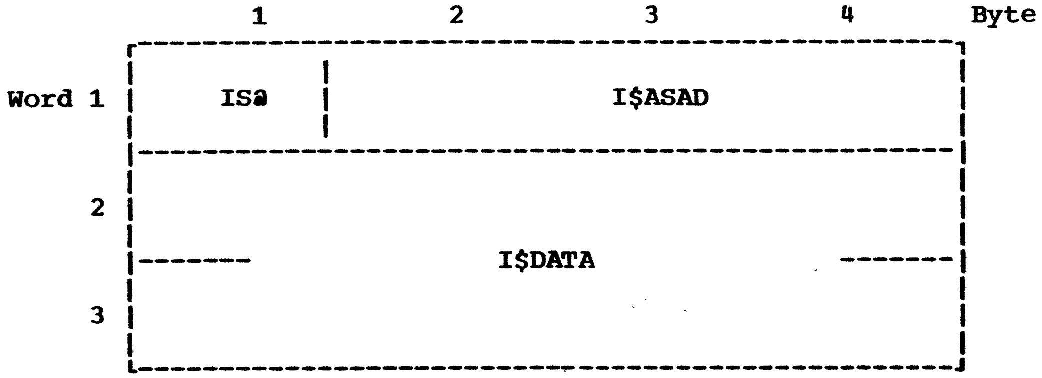
where I$NODE is less than 20 and indicates the number of bytes of data in I$DATA to go into constant storage. I$ASAD is the offset into constant storage.

2. Adcon initialization node

```
                1           2           3           4        Byte
         r-----------------------------------------------------1
         |            |                                        |
  Word 1 |    IA@     |               I$ASAD                   |
         |-----------------------------------------------------|
         |                                                     |
       2 |                 .     I$DATA                        |
         |                                                     |
         L-----------------------------------------------------J
```

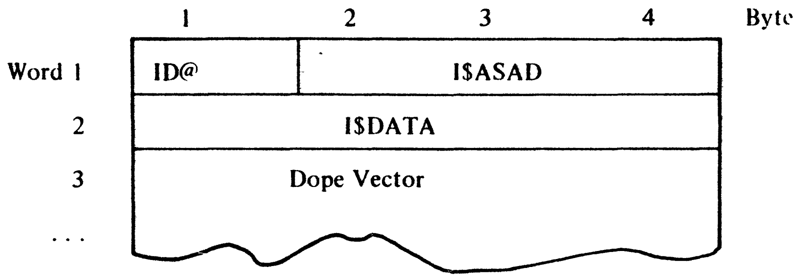
where I$ASAD is the offset in the adcon area to place the adcon indicated by I$DATA. This word contains a one-byte base code and three-byte displacement.

3. SDV initialization node



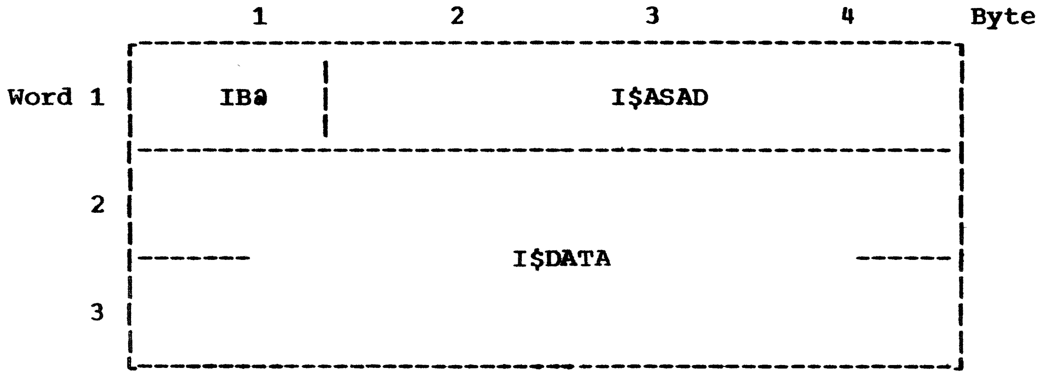
where I$ASAD is offset in constant storage for the string dope vector. If the first word of I$DATA is nonzero, the offset to static arrays storage needs to be added to the rightmost three bytes of the first word.

4. ADV/SADV initialization node



where I$ASAD is offset in constant storage for the dope vector. I$DATA is the length of the dope vector in bytes. The dope vector starts at I$DATA+4.

5. BAA initialization node



where I$ASAD is offset in the adcon area for the block adcon area. The first word of I$DATA contains the address of the block's entry point. The byte at I$DATA+4 contains the number of parameters for the block.

6. Special SDV entry

	1	2	3	4	Byte
Word 1	IDV@	I$ASAD			
2	I$DATA				
3					

where I$ASAD is offset in constant storage for the dope vector. The offset to static storage needs to be added to the rightmost three bytes of the first word of I$DATA.

7. Discarded entry

	1	2	3	4	Byte
Word 1	X' 28'	Not Used			
2	I$DATA				
. . .	Not Used				

This is a dope vector entry that has been discarded and thus needs no initialization. I$DATA has the number of bytes in the dope vector.

TITLE: DOPE VECTOR LIST (J LIST)

Purpose and Usage

The dope vector list (also called supplementary initialization list) is generated by the Attribute Node Creation subroutine ($ANCRE) for skeletal dope vectors that are too large to fit into an initialization table (I table) segment. Compilation Wrap-Up Driver ($MCWU) places the dope vector in static storage.

Description and Entry Format

The J list is stored as a true list; that is, the first word of each entry in the list points to the next entry. Except for the extra link word, an entry looks exactly the same as an ADV/SADV initialization node in the I table.

	1	2	3	4	Byte
Word 1	J$NODE	J$NEXT			
2	J$LOC				(Same as I$ASAD)
3	J$CNT				(Same as I$DATA)
4	J$DATA				(Dope Vector)
• • •					

TITLE: LIBRARY LOAD TABLE (L TABLE)

Purpose and Usage

The library load table is used to record which library runtime routines will be needed during execution of a program. Entries are made in this table by the Library Search routine ($NLSIB). At wrap-up time, the Runtime Library Loader routine ($HRTLL) uses this table to determine which runtime routines must be loaded.

Description

The library load table is loaded into the fixed area of working storage. At the start of compilation, this table is set to zeros. A number which maps to a unique word in the library load table is associated with each runtime library routine. If a particular library routine is needed at object time, its word in the library load table is set to point to a word in the adcon area. (See "Phase 2 Initializer ($WCONT)" in Section 3, Volume I.) At runtime, this word in the adcon area will contain the location of the library routine.

Entry Format

The format of each entry in the library load table is shown below.

```
               1       2       3       4    Byte
             ┌────────────────────────────────┐
     Word 1  │   Adcon Displacement or Zero   │
             └────────────────────────────────┘
```

If nonzero, this word will contain a displacement from the start of the adcon area. At runtime, the adcon location that is pointed to will contain the location at which the library routine corresponding to this entry has been loaded.

If zero, either loading of the routine which corresponds to this entry has not been requested or no routine corresponds to this entry. (There are more words in the library load table than there are runtime library routines.)

TITLE: SYMBOLIC INSTRUCTION TABLE (M TABLE)

Purpose and Usage

The symbolic instruction table is used by the Triad Code Generator ($TCODE) to communicate with the Instruction Assembler ($VINSA). The number of instructions and the origin of the instruction sequence in the symbolic instruction table is selected by $TCODE and then processed by $VINSA. This instruction sequence does not provide for covering of operand addresses. Instructions to provide cover are generated by $VINSA. The instruction sequence may contain loads of registers which are discarded by $VINSA if the operand is already in an appropriate register. The instruction may be modified by $VINSA to select the instruction appropriate to the operand type (fixed, single float, or double float) or to change from the RX to the RR form of the instruction if the operand is in an appropriate register.

Description

The table is of fixed length and all entries contain preset values, each representing a symbolic machine instruction or pseudo-instruction, which are never modified by the compiler. The table is actually assembled as part of the $TCODE module. The format of the symbolic instruction table is as follows:

```
 1         9      13     17        25        33     37     41       49       56 Bit
r-----------------------------------------------------------------------------------1
|          |      |      |         |         |      |      |        |          |
|  INSTNO  |TOP1  |TOP2  |  VOP1   |  VOP2   |TOP3  |TOP4  |  VOP3  |   VOP4   |
|          |      |      |         |         |      |      |        |          |
L-----------------------------------------------------------------------------------J
```

Values for Fields:

INSTNO Instruction Number. This value, when multiplied by 2, is used to index into the operation code table (O table) to obtain the machine instruction code and the operation characteristics mask.

TOP1 and VOP1 The TOP1 field value indicates the type of information contained in the VOP1 field. TOP1 can assume any value from 0 through 7. See Figure B-9 for corresponding value of VOP1 field.

TOP2 and VOP2 The TOP2 field value indicates the type of information contained in the VOP2 field. TOP2 can assume any value from 0 through 7. See Figure B-9 for corresponding value of VOP2 field.

TOP3 and VOP3 The TOP3 field value indicates the type of information contained in the VOP3 field. TOP3 can assume a value of 1, 2, or 3. See Figure B-9 for corresponding value of VOP3 field.

TOP4 and VOP4 The TOP4 field value indicates the type of information contained in the VOP4 field. TOP4 can assume a value of 1, 2, or 3. See Figure B-9 for corresponding value of VOP4 field.

Note: TOP3, TOP4, VOP3, and VOP4 are present only for an instruction whose operation code indicates indexing. VOP3 generates an X1 field ; VOP4, a B2 field.

TOP	VOP
0 (NULL)	Null operand. Has no effect on assembled instruction.
1 (SCR)	Symbolic computational register. Symbolic registers are labeled 0, 2, 4, and 6. An odd number value indicates the low-order half of a symbolic register pair, if it is a double register. The assigned register will be fixed or floating-point, depending upon the data requirements.
2 (SAR)	Symbolic adcon register. Same meaning as above except register assigned is a general adcon register.
3 (ABSR)	Absolute register. Value is the displacement into the register table for the entry associated with the absolute register.
4 (PARM)	Data parameter. Value of operand is 0, 12, 24, 36, 48, or 60, indicating the relative distance into the data parameter table ($PTO table) of the information pertaining to the desired operand.
5 (REAL)	Real address of data parameter. Value is as for type 4, except reference applies to real part only.
6 (IMAG)	Imaginary address of data parameter. Value is as for type 4, except that addresses are to be adjusted for the imaginary part of complex data.
7 (SSTG)	Scratch storage. Where 0 and 4 means entire value, 1 and 5 means real part, and 2 and 6 means imaginary part of the first and second scratch work areas, respectively. Scratch storage is reused on each call to $VINSA.
8 (CONS)	Constant entry pointer. Relative address within the constant entry portion of the operation code table which contains the operand value.
9 (LIT)	Literal value. The operand itself.
10 (LBL)	Symbolic label. Symbolic labels are numbers 0,1,2,...,9. Only one instruction may branch to a symbolic label. An instruction may have more than one symbolic label.

Figure B-9. Operand Values for Symbolic Instruction Table

TITLE: DICTIONARY NAME LIST (N LIST)

Purpose and Usage

The dictionary name lists constitute a central depository in the compiler for each distinct identifier and a pointer to its associated definition list. Combined with the dictionary hash table, these lists provide the means through which an identifier or information about the identifier may be referenced.

Description

The dictionary hash table is an ordered table of pointers to dictionary name lists. Each dictionary name list in turn is composed of one entry for each distinct identifier that hashes to the same value. This includes identifiers in the source program as well as built-in function names, syntactic keywords, etc.

The list nodes are of standard list structure format; thus the first word of each contains the node type and a pointer to the next node of the list. Each list may contain two types of nodes, one type for four-character identifiers and another for eight-character identifiers. The name entries in each list are ordered in sorting order from low to high with four-character identifiers preceding eight-character identifiers.

Each node contains a keyword-type flag and a pointer to the definition list of the identifier. The keyword-type flag provides an indication of whether the identifier is a potential keyword, and, if so, a unique identification of the keyword. The dictionary name lists are initialized with entries for all keywords, built-in function names, etc.

The third and, where applicable, fourth words of each node contain the identifier in EBCDIC. All identifiers are filled out with blanks until they are either four or eight characters in length.

Entry Formats

The formats for entries in a dictionary name list are shown below.

<table>
<tr><td>OA</td><td>Pointer to next entry in list</td></tr>
<tr><td>Keyword Type</td><td>Pointer to definition list</td></tr>
<tr><td colspan="2">Identifier (4 characters)</td></tr>
</table>

<table>
<tr><td>OB</td><td>Pointer to next node in list</td></tr>
<tr><td>Keyword Type</td><td>Pointer to definition list</td></tr>
<tr><td colspan="2">Identifier (8 characters)</td></tr>
</table>

Values for N$KEY (Keyword Type):

N@NULL	0	NOT A KEYWORD
N@DCL	1	DECLARE (DCL)
N@FMT	2	FORMAT
N@ELSE	3	ELSE
N@END	4	END
N@PROC	5	PROCEDURE (PROC)
N@BGN	6	BEGIN
N@IF	7	IF
N@ON	8	ON
N@DO	9	DO
N@RET	10	RETURN
N@CALL	11	CALL
N@GET	12	GET
N@GO	13	GO
N@GOTO	14	GOTO
N@PUT	15	PUT
N@RVT	16	REVERT
N@STOP	17	STOP
N@OPEN	18	OPEN
N@CLOSE	19	CLOSE
N@STMT	20	END OF STATEMENTS

Condition Keywords

N@ERR	22	ERROR
N@FOFL	23	FIXEDOVERFLOW (FOFL)
N@OFL	24	OVERFLOW (OFL)
N@UFL	25	UNDERFLOW (UFL)
N@ZDIV	26	ZERODIVIDE (ZDIV)
N@ENDF	27	ENDFILE

Filenames

N@INFL	29	INPUT FILE
N@SYIN	30	SYSIN
N@ANYF	31	ANY FILE
N@SYPT	32	SYSPRINT
N@OTFL	33	OUTPUT FILE

Options (Non I/O)

N@TO	40	TO
N@BY	41	BY
N@WHLE	42	WHILE
N@THEN	43	THEN
N@OPTN	44	OPTIONS
N@SYTM	45	SYSTEM
N@INTN	46	INTERNAL (INT)
N@EXTN	47	EXTERNAL (EXT)

FORMAT Specs

N@R	53	R
N@C	54	C
N@F	55	F
N@E	56	E
N@A	57	A
N@X	58	X
N@COLM	59	COLUMN (COL)
N@SKIP	60	SKIP

GET/PUT Options (Plus SKIP)

N@EDIT	61	EDIT
N@LIST	62	LIST
N@DATA	63	DATA
N@FILE	64	FILE

Attributes (Plus FILE)

N@INPT	65	INPUT
N@OTPT	66	OUTPUT
N@PNT	67	PRINT
N@STIC	68	STATIC
N@AUTO	69	AUTOMATIC (AUTO)
N@LBL	70	LABEL
N@ENV	71	ENVIRONMENT (ENV)
N@RETS	72	RETURNS
N@CHAR	73	CHARACTER (CHAR)
N@ENTY	74	ENTRY
N@FXD	75	FIXED
N@FLT	76	FLOAT
N@CPLX	77	COMPLEX (CPLX)
N@REAL	78	REAL
N@TITLE	79	TITLE

TITLE: OPERATION CODE TABLE (O TABLE)

Purpose and Usage

The operation code table is used by the Instruction Assembler ($VINSA) to interpret the symbolic instruction table (M table).

Description

The table is of fixed length and all entries contain preset values which are never modified by the compiler. The first part of the table contains constant entries which are constant operands greater than a byte in length. The second part of the table contains operation entries, each representing a machine operation or pseudo-operation. This table is actually assembled as part of the $TCODE module.

Entry Formats

The format of constant entries in the operation code table is detailed below.

1	2 N+1 Byte
LENGTH	VALUE

Values for Fields:

LENGTH Number of bytes, N, of VALUE.

VALUE The N bytes representing the binary value of the constant.

The format of operation code entries in the table follows.

1	2	3	4	5 6 7 8	9 16 Byte
M	R	S	X	EFF	OPCODE

Values for Fields:

M If 1, the operation code modifier is to be added to the OPCODE field. This adjusts the instruction for fixed, single float, or double float.

R If 1, the OPCODE field is modified by -X'40' if operand is in a register.

S If 1, the OPCODE field is a pseudo-operation number.

X If 1, the generated instruction is to be indexed with an X1 and/or B2 field.

EFF Defines the effect of the operation on the register where:

0 = Loads the register positively with new value.
1 = Loads the register negatively with new value.
2 = Destroys the register value, result single register.

3 = Destroys the register value, result of a fixed multiply.
4 = Destroys the register value, result of a fixed divide.
5 = Stores the register value.
6 = Destroys all register synonyms.
7 = Has no effect on register value.
8 = Changes sign of register (inverts).
9 = End of procedure.

OPCODE The value in this field is dependent on the value in M, R, S, or X as described above.

TITLE: PROGRAM STRUCTURE TABLE (P TABLE)

Purpose and Usage

The program structure table describes the program structure statement currently in effect. It contains one entry for each currently unterminated BEGIN, PROCEDURE, IF, DO, or ON statement.

Description

The program structure table is an expandable table with six different types of entry. Each entry is four words long. The table is treated like a push-down list.

Entry Formats

Two bytes of the first word of the entry and the last word of the entry contain standard information. The information contained in the rest of the entry is dependent on the type of entry.

The first byte of the first word contains a node type. This describes which of the six entry types is represented. The last byte of the first word contains the value of the compound statement switch at the time the entry was made.

The last word of the entry contains a pointer to the dictionary name list (N list) entry for the label on the statement. If there was no label or if the statement is an IF or ON statement, this pointer is null.

Values for general fields:

P$TYPE:	P@NIDO	Noniterative DO
	P@IDO	Iterative DO
	P@IF	IF statement
	P@ON	ON statement
	P@BEG	BEGIN statement
	P@PROC	PROCEDURE statement
P$CSS:	Values for $CSS	

Noniterative DO Statement Entry: The entry for a noniterative DO contains no information other than the general information.

The format for a noniterative DO entry is shown below:

```
                1           2         3         4         Byte
          r-----------------------------------------------------1
   Word 1 | Node Type |                        | Saved CSS  |
          |  P$TYPE   |       Not Used         |   P$CSS    |
          |-----------------------------------------------------|
        2 |                                                     |
          |                                                     |
          |------------------  Not Used  -----------------------|
        3 |                                                     |
          |                                                     |
          |-----------------------------------------------------|
        4 |         Pointer to Name Entry of Label              |
          |                  P$LNEP                             |
          L-----------------------------------------------------J
```

Noniterative DO Statement Entry

Iterative DO Statement Entry: An iterative DO entry is created for a DO statement that contains a TO, BY, or WHILE clause. Besides the general information, this entry contains a nonrepeating switch. This switch, contained in bit 0 of the second byte of the first word, is 0 if a branch to the increment and test instructions should be generated at the end of the DO. If the bit is 1, branch instructions are not to be generated. The second word of an entry contains the address of the increment and test instructions. The third word contains the address of the forward internal branch to the end of the DO-loop. This branch needs to be resolved at the end of the DO.

The format of an iterative DO entry is shown below:

	1	2	3	4	Byte
Word 1	Node Type P$TYPE	Nonrepeating Switch P$NRSW	Not Used	Saved CSS P$CSS	
2	Address of Increment and Test Instructions P$ITA				
3	Address of Branch to End of Loop Instructions P$BELA				
4	Pointer to Name Entry of Label P$LNEP				

Iterative DO Statement Entry

Values for Fields:

P$NRSW = 0 Generate branch back to DO.
= X'80' Do not generate branch back to DO.

IF Statement Entry: An IF entry is created for an IF statement in the source program. The second word of the entry contains the address of the forward internal branch to the ELSE clause. The third word contains the address of the forward internal branch to the end of the IF statement. Both of these branches need to be resolved when the branch point is reached.

The format of an IF entry is shown below:

	1	2	3	4	Byte
Word 1	Node Type P$TYPE	Not Used		Saved CSS P$CSS	
2	Address of Branch to ELSE Clause P$BECA				
3	Address of Branch to End of IF P$BEFA				
4	Null				

IF Statement Entry

ON Statement Entry: An ON entry is created for an ON statement in the source program. The second word contains the address of the forward internal branch to the end of the on-unit. This branch needs to be resolved at the end of the unit. The third word contains the address of the on-unit adcon area in adcon storage. This adcon area is very similar to a BAA and contains information about the on-unit. (See Appendix E.)

The format of an ON entry is shown below:

	1	2	3	4	Byte
Word 1	Node Type P$TYPE	Not Used		Saved CSS P$CSS	
2	Address of Branch to End of On-Unit P$BEUA				
3	Address of On-Unit Adcon Area P$ADCA				
4	Null				

ON Statement Entry

<u>BEGIN</u> <u>Statement</u> <u>Entry</u>: A BEGIN entry is created for a BEGIN statement in the source program. The second byte of the first word contains the block number of the begin block. The second word contains a pointer to the block information table (B table) entry for the begin block.

The format for a BEGIN entry is shown below:

	1	2	3	4	Byte
Word 1	Node Type P$TYPE	Block Number P$BLNR	Not Used	Saved CSS P$CSS	
2	Pointer to Block Information Table Entry P$BITP				
3	Not Used				
4	Pointer to Name Entry of Label P$LNEP				

BEGIN Statement Entry

<u>PROCEDURE</u> <u>Statement</u> <u>Entry</u>; A PROCEDURE entry is created for a PROCEDURE statement in the source program. As for a BEGIN entry, the second byte of the first word contains the block number of the block and the second word contains a pointer to the block information table (B table) entry for the block. The third word of the entry contains the address of the forward internal branch around the procedure block. This branch needs to be resolved at the end of the block.

The format of a PROCEDURE entry is shown below:

	1	2	3	4	Byte
Word 1	Node Type P$TYPE	Block Number P$BLNR	Not Used	Saved CSS P$CSS	
2	Pointer to Block Information Table Entry P$BITP				
3	Address of Branch to End of Block P$BEBA				
4	Pointer to Name Entry of Label P$LNEP				

PROCEDURE Statement Entry

TITLE: SUBSCRIPT SUBSTITUTION TABLE (Q TABLE)

Purpose and Usage

The subscript substitution table is used to create a token list for the subscripts generated for an array expression.

Description and Entry Format

Every segment of this table, except the first, looks exactly like a token table (T table) segment. The first segment contains all tokens; no space is reserved at the beginning of this segment for line position. The table is kept in this manner so that it can be processed using the Get Token macro (GETKN) (see Appendix C).

The tokens placed in this table consist of a left parenthesis followed by the subscript designators separated by commas and followed by a right parenthesis.

TITLE: REGISTER TABLE (R TABLE)

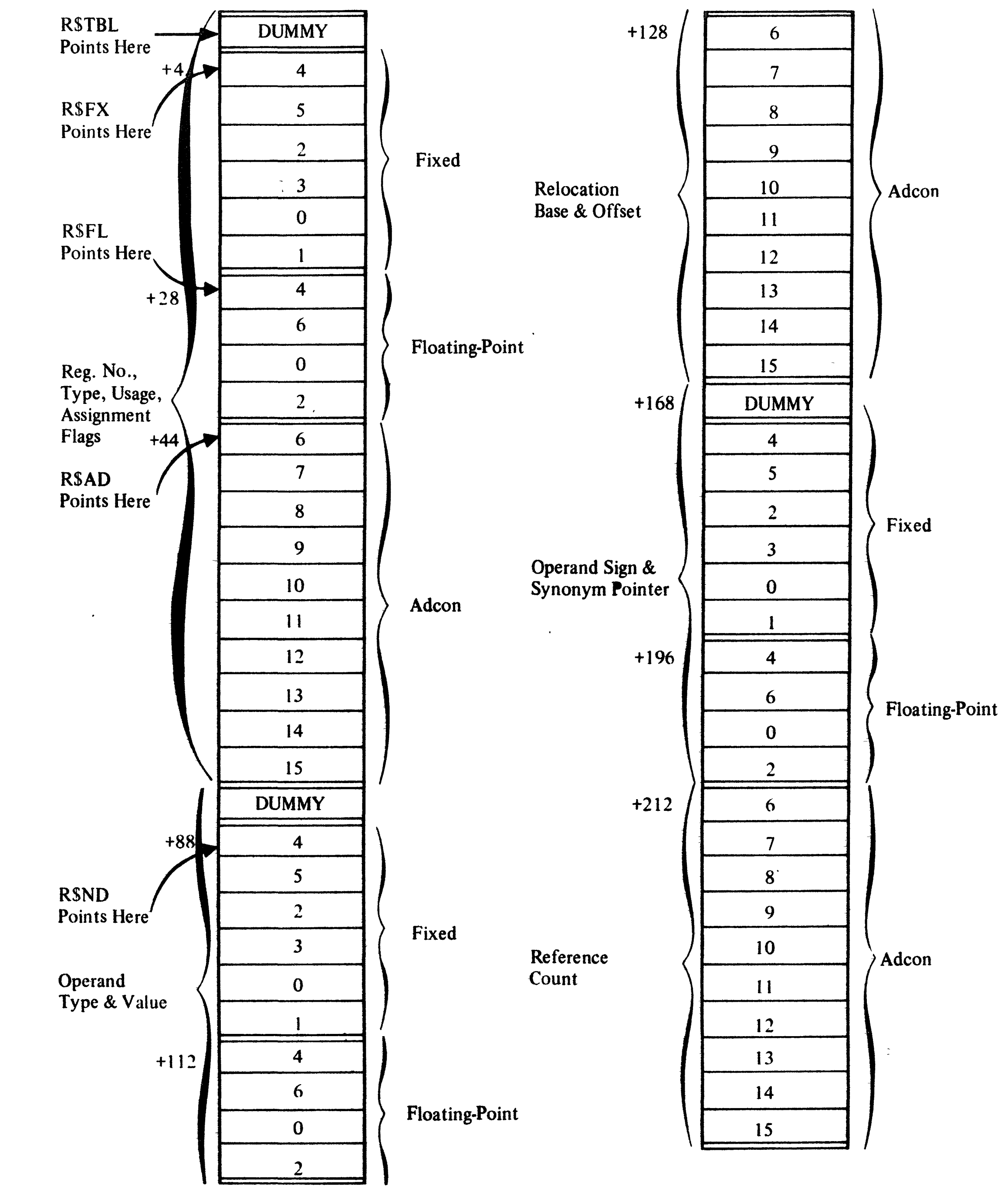
Purpose and Usage

The register table is used by the Instruction Assembler routine ($VINSA) to maintain the current status of all registers in the object program.

Description

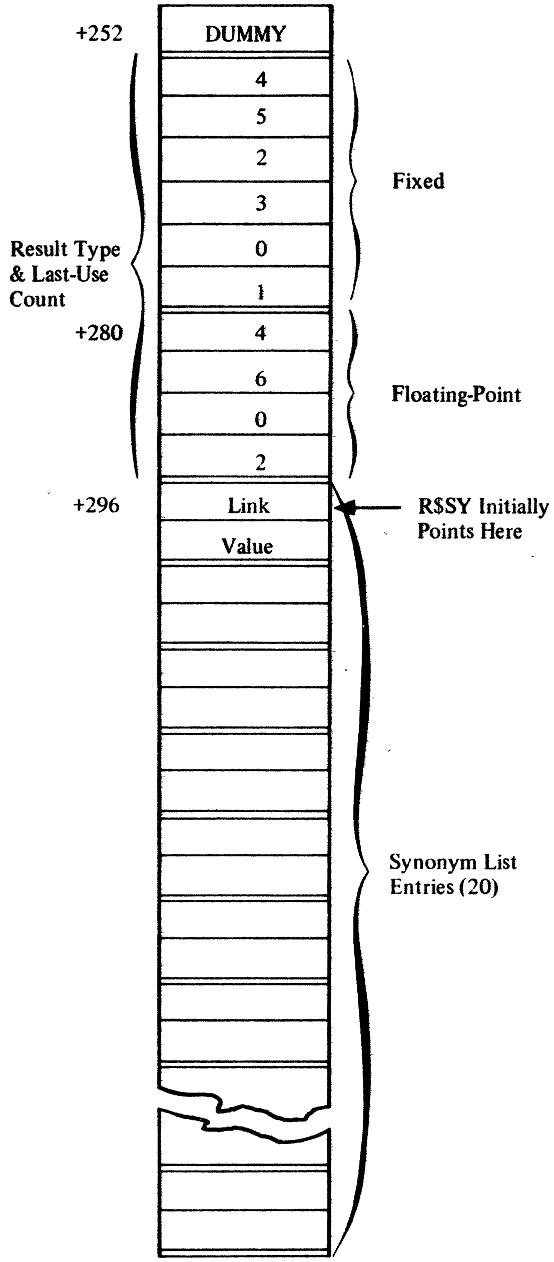
This table is of fixed length and is divided into four parts: adcon register table, fixed register table, floating-point register table, and synonyms. The adcon section contains one entry for each of the ten general registers assigned as nonpermanent address constant registers. The fixed register section contains one entry for each of the six general registers assigned as computational registers. The floating-point register section contains one entry for each of the four floating-point computational registers. The synonym section contains 20 entries.

The general structure of the register table is shown in Figure B-10.



R Table (1 of 2)

Figure B-10. Format of Register Table (Page 1 of 2)



R Table (2 of 2)

Figure B-10. Format of Register Table (Page 2 of 2)

Entry Formats

The computational registers, both fixed and floating-point, have the following format:

```
                1            2            3            4       Byte
         ┌─────────────────────────────────────────────────────┐
  Word 1 │ REG NO        │ REG TYPE  │ GLOB IND    │ USE IND  │
         │               │           │             │          │
         │───────────────┼──────────────────────────────────────│
       2 │ OP TYPE       │             OP VALUE                 │
         │               │                                      │
         │───────────────┼──────────────────────────────────────│
       3 │ OP SIGN       │             NEXT SYN                 │
         │               │                                      │
         │───────────────┼──────────────────────────────────────│
       4 │ RESULT TYPE   │             LAST-USE                 │
         │               │                                      │
         └─────────────────────────────────────────────────────┘
```

The adcon registers have the following entry format:

```
                1            2            3            4       Byte
         ┌─────────────────────────────────────────────────────┐
  Word 1 │ REG NO        │ REG TYPE  │ GLOB IND   │           │
         │               │           │            │           │
         │───────────────┼──────────────────────────────────────│
       2 │   BASE        │                OFFSET                │
         │               │                                      │
         │───────────────┼──────────────────────────────────────│
       3 │               │                REF NO                │
         │               │                                      │
         └─────────────────────────────────────────────────────┘
```

Values for Fields:

REG NO — Absolute register number associated with entry.

REG TYPE — Type of register; 0 = Adcon, 1 = Fixed computational, 2 = Floating-point computational

GLOB IND — When ON, the following bits signify:

Bit 7: The register is globally assigned to the indicated value.
Bit 6: The register is symbolically assigned to the indicated value.
Bit 5: The register is inhibited from assignment.

USE IND — If 0, assigned as a single register; if 1, assigned as left half of a double register; if 2, assigned as right half of a double register.

OP VALUE — Dictionary attribute list pointer, constant table pointer, etc.

OP TYPE — Indicates the type of the op value field.

OP SIGN — If 1, the operand has a prefix minus sign.

NEXT SYN — If nonzero, points to the first synonym entry which applies to the register.

RESULT TYPE — The bits have the following values and meanings when ON:

Bit 0 = Result of a fixed multiply
Bit 1 = Result of a fixed divide
Bit 3 = Double precision result
Bit 6 = Floating-point result
Bit 7 = Complex result

LAST-USE — Number of the triad after which the contents of the register can be destroyed.

BASE — Relocation base of the adcon.

OFFSET — Relative address of adcon from base origin.

REF NO — Binary number associated with the referencing of the adcon register; the lower the value, the more distant the last reference is from the current processing point.

Synonyms have the following format:

```
                1           2           3           4    Byte
         ┌───────────┬─────────────────────────────────────┐
  Word 1 │           │       Pointer to Next Synonym       │
         │           │                                     │
         ├───────────┼─────────────────────────────────────┤
       2 │  Op Type  │            Op Value                 │
         │           │                                     │
         └───────────┴─────────────────────────────────────┘
```

TITLE: TEMPORARY STORAGE TABLE (S TABLE)

Purpose and Usage

Each procedure block, format list, and on-unit has temporary storage associated with it. Each level of temporary storage is identified by an entry in the temporary storage table.

Description

The temporary storage table is maintained as an expandable table. It contains one logical section for each unterminated procedure block, on-unit, and format list. This table is referenced like a push-down list.

Entry Format

Each entry in the table is three words long. The first word contains the length in bytes of the temporary storage area, and the number of the triad after which the area will be available for reuse (last-use count). The second word contains the base code and displacement of the storage location allocated to the temporary area. The third word contains the DO level associated with the temporary area when it is used in conjunction with register storage around DO-loops. An entry of zero in the first word indicates the beginning of a logical section of subsequent entries and separates the entries for one block from those for the encompassing block.

The format of an entry in the temporary storage table is shown below.

```
              1           2           3          4       Byte
          ┌──────────────────────────────────────────────┐
  Word 1  │      Length          │    Last-Use Count     │
          │──────────────────────────────────────────────│
       2  │      Allocated Base Code and Displacement    │
          │──────────────────────────────────────────────│
       3  │                    Level                     │
          └──────────────────────────────────────────────┘
```

TITLE: TOKEN TABLE (T TABLE)

Purpose and Usage

The token table contains one entry for each token in a source statement. The token table also contains an attribute entry for each occurrence of a constant in the statement.

Description

The token table is maintained as an expandable table. The Entoken routine ($ATKN) entokens one statement at a time and places it in the token table. At the beginning of entokening, all areas in this table except the first are released. Thus the space used for the token table is statement-related.

A segment of the token table is subdivided as shown in Figure B-11.

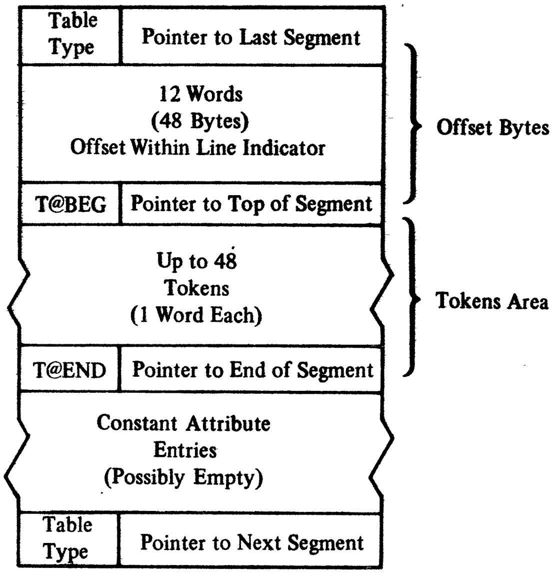


Figure B-11. Format of Token Table

The offset bytes are used to indicate the character position on the line where the corresponding token in the tokens area began. This information is used only in producing error messages. The number of bytes in the area is fixed; there are enough bytes for the maximum number of tokens in the tokens area.

Entry Format

All token table entries are one word in size. T$TYPE is used to identify the type of token. The following types are used (not including end-of-table type):

identifier
constant
non-parenthesis delimiter
right-parenthesis delimiter
left-parenthesis delimiter
keyword
new line.

T$PTR of an identifier entry points to the dictionary name list (N list) entry for the identifier. The format of an identifier entry is:

T$TYPE = T@ID 08	T$PTR

T$PTR of a constant entry points to a constant attribute (A list) entry. The format of a constant entry is:

T$TYPE =T@CNST 0C	T$PTR

T$PRTY of a non-parenthesis delimiter entry contains the priority of the delimiter. The priority is a combination of a one-byte parenthesis count followed by the absolute priority of the delimiter in one byte. The format of a non-parenthesis delimiter entry is:

T$TYPE = T@DELM 18	T$DELM	T$PRTY	
		Parenthesis Level	Absolute Precedence

In addition, the token for a right-parenthesis delimiter contains in T$COM the count of the number of commas inside the parentheses pair not included inside a contained parentheses pair. The format of a right-parenthesis delimiter entry is:

T$TYPE =T@RPR 14	T$COM	T$PRTY

T$PTR of a left-parenthesis delimiter entry contains a pointer to the corresponding right-parenthesis entry. This pointer is always valid since the Entoken routine ($ATKN) balances parentheses. The format of a left-parenthesis delimiter entry is:

T$TYPE =T@LPR 10	T$PTR

If an identifier longer than eight characters appeared in the source and was a legal keyword, then a keyword token is inserted in the token table. T$KEY of this entry contains the keyword type which is normally contained in the dictionary name list (N list) entry. (See "Dictionary Name List (N List)" for keyword codes.)

The format for a keyword entry is:

T$TYPE =T@KEY 04	T$KEY	Not Used

A new line entry signifies that a new line in the source program began with the next token. T$PTR of the entry points to the beginning of the line number in the source program. The format for a new line entry is:

T$TYPE =T@NEW 20	T$PTR

If a new line begins in the middle of a character string, then the character string constant tokens follow the new line token. Multiple new line tokens in a row are reduced so that only the last one is present in the token table.

T$PTR for a current end or beginning of table token points to the end or beginning of the current area. The end of table token entry does not necessarily appear at the physical end of an area. The format for this entry is:

Token Type	Pointer to Next Area

TITLE: EXPRESSION STACK (V TABLE)

Purpose and Usage

The first entry of the expression stack table is created by one of the statement processing routines to indicate the type of expression to be processed. All remaining entries are added by the Expression Processor Controller ($NEXP). Each time the start of a new expression is detected (that is, a left parenthesis is encountered), the stack is pushed down and an entry created for the new expression. Each time the end of an expression is detected (that is, a right parenthesis is encountered), the stack is popped up and the top entry removed.

Description

The expression stack is maintained as an expandable table and processed as a push-down list.

Entry Format

The format of an entry in the expression stack is shown below.

```
                  1          2          3          4        Byte
           ┌──────────────────────────────────────────────┐
    Word 1 │   V$TYPE   │ V$CCNT   │  V$ACNT   │  V$PTYP    │
           │            │          │           │            │
           │──────────────────────────────────────────────│
           │                                              │
         2 │              V$ATTE or V$CSDV                │
           │──────────────────────────────────────────────│
           │                                              │
         3 │                   V$PRMP                     │
           └──────────────────────────────────────────────┘
```

Values for Fields:

V$TYPE The type of expression:

Value	Symbolic Name	Meaning
0	V@ARG	Argument list
4	V@SUBL	Subscript list
8	V@ACRS	Array cross-section
12	V@EXP	Ordinary expression
16	V@ASS	Assignment statement
20	V@AASS	Array assignment statement

V$CCNT The comma count if a list is under process.

V$ACNT The * count if a cross-section is under process.

V$PTYP The type of the next outer expression.

V$ATTE Pointer to the array or entry name if a list is under process.

V$CSDV Offset in static storage to the origin of the cross-section dope vector if an array cross-section is under process.

V$PRMP Pointer to the dictionary attribute node of the next parameter if the attributes of the arguments are declared.

TITLE: OPERATOR STACK (X TABLE)

Purpose and Usage

The operator stack is created by the Expression Processor Controller ($NEXP) to hold operators which have not been processed to form triads. One entry is active for each operator whose operands are still indeterminate.

Description

The operator stack is maintained as an expandable table and processed as a push-down list. Each operator and separator encountered in the token table (T table) by the Expression Processor Controller is added to the top of this stack. Before adding the new operator, all operators at the top of the stack for which the new operator determines operands are processed and removed from the stack.

Entry Format

The format of an entry in the operator stack is shown below.

```
              1          2          3          4    Byte
        ┌──────────────────────────────────────────┐
        |            |          |        X$PRTY     |
Word 1  |            |  X$DELM  |- - - - -|- - - - -|
        |            |          | X$PRNL  | X$PRCL  |
        |------------+------------------------------|
        |            |                              |
     2  |   X$POI    |            X$TOKN            |
        |            |                              |
        └──────────────────────────────────────────┘
```

Values for Fields:

X$DELM: Delimiter type:

Value	Symbolic Name	Meaning
28	T@COM	Comma
32	T@ASGN	Assignment symbol
36	T@DIV	Divide
40	T@MIN	Infix minus
44	T@EXP	Exponentiation
48	T@GT	Greater than
52	T@GTE	Greater than or equal
56	T@LT	Less than
60	T@LTE	Less than or equal
64	T@EQ	Relational equal
68	T@NE	Not equal
72	T@OR	Logical OR
76	T@AND	Logical AND
80	T@MPY	Multiply
84	T@PLS	Infix plus
88	T@LEFT	Left parenthesis

X$PRTY The priority of the operator which consists of a parenthesis level and a precedence number of the operator.

X$PRNL The most significant part of the operator priority, the level of nesting of parenthesis at which the operator occurred.

X$PRCL The least significant part of the operator priority, the precedence number assigned to the operator.

X$POI If bit 6 is 1, a prefix operator is applied to the delimiter. If bit 7 is 1, an odd number of prefix minus operators is applied to the delimiter.

X$TOKN Pointer to the token table entry which contains the delimiter token.

TITLE: OPERAND STACK (Y TABLE)

Purpose and Usage

The operand stack is created by the Expression Processor Controller ($NEXP) to hold operands which have not been processed to form triads. One entry is active for each such operand.

Description

The operand stack is maintained as an expandable table and processed as a push-down list. Each operand encountered in the token table (T table) by the Expression Processor Controller is added to the top of this stack. Whenever it is possible to combine an operator with its two operands, the topmost entries of the stack representing the operands are removed and replaced with an entry indicating the result of the operation.

Entry Formats

An identifier or source constant has the following format:

```
                 1           2           3           4       Byte
          ┌──────────────────────────────────────────────────┐
          │           │                                      │
   Word 1 │  Y$OPT    │               Y$OPV                  │
          │───────────┼──────────────────────────────────────│
          │           │                                      │
        2 │  Y$POI    │               Y$TOKN                 │
          └──────────────────────────────────────────────────┘
```

Values for Fields:

Y$OPT The type of operand:

Value	Hex	Symbolic Name	Meaning
12	C	T@CNST	Pointer to attribute node of a source constant
28	1C	T@IDAT	Pointer to attribute node of an identifier
40	28	T@PSEU	Pointer to attribute node of a pseudo-variable

Y$OPV Meaning dependent on value of Y$OPT.

Y$POI If bit 0 is 1, the operand is parenthesized. If bit 6 is 1, a prefix operator is applied to the operand. If bit 7 is 1, an odd number of prefix minus operators is applied to the operand.

Y$TOKN Pointer to the token table entry which contains the operand.

All other entries have the following format:

```
                 1          2           3           4    Byte
          ┌──────────────────────────────────────────────┐
          |           |                                  |
   Word 1 |   Y$OPT   |              Y$OPV               |
          |           |                                  |
          |-----------+----------------------------------|
          |           |           |         Y$L          |
        2 |   Y$POI   |   Y$TM    |- - - - - -|- - - - - |
          |           |           |   Y$PR    |   Y$SF   |
          └──────────────────────────────────────────────┘
```

Values for Fields:

Y$OPT The type of operand:

Value	Hex	Symbolic Name	Meaning
44	2C	T@TRID	Pointer to triad table (Z table) entry
48	30	T@DATA	Immediate constant
52	34	T@LIB	Entry point number of library routine
56	38	T@ERR	Error operand
60	3C	T@HADD	Halfword address in static storage
64	40	T@FADD	Fullword address in static storage
68	44	T@BADD	Byte address in static storage
72	48	T@PADD	Immediate DED
76	4C	T@IBIF	In-line built-in function number
80	50	T@CON	Pointer to constant table (C table)

Y$OPV: Meaning dependent on value of Y$OPT.

Y$POI: Same meaning as above.

Y$TM: Operand type mask. Has same meaning as dictionary attribute data description field:

Bit 0	If 1, operand is arithmetic.
Bit 1	If 1, operand is a variable.
Bit 2	If 1, operand is in error.
Bit 3	If 1, operand is double precision.
Bit 4	If 1, operand is of special type.
Bit 5	If 1, operand is a character string.
Bit 6	If 1, operand is floating-point.
Bit 7	If 1, operand is complex.

Y$L: The length if operand is a string.

Y$PR: The precision of the operand if its type is arithmetic.

Y$SF: The scale factor of the operand if its type is arithmetic.

TITLE: TRIAD TABLE (Z TABLE)

Purpose and Usage

The triad table contains one entry for each triad (that is, operator and two operands) generated by the statement processors and the Expression Processor Controller ($NEXP). The triads are ordered such that they appear in the triad table in object program execution order. The triad entries are processed by the Triad Code Generator ($TCODE) to produce machine-language instructions.

Table Description

The triad table is maintained as an expandable table and contains all the triads required to represent a single source statement. Each entry of the table is four words in length.

Entry Formats

The format of each triad before code is generated is:

	1	2	3	4 Byte
Word 1	Z$TOP	Z$LOS	Z$ROS	Z$TMSK
2	Z$LOPT	Z$LOP		
3	Z$ROPT	Z$ROP		
4	Z$TSGN	Z$LUSE		

Values for Fields:

Z$TOP Indicates the specific operation to be performed on the operands. The operators are:

Value	Hex	Symbolic Name	Meaning
0	0	T@NULL	Null
4	4	T@LAST	End of triad table
8	8	T@CALL	Begin call
12	C	T@ENDC	End call
16	10	T@CVT	Convert
28	1C	T@COM	Argument list (comma)
32	20	T@ASGN	Assignment symbol
36	24	T@DIV	Divide
40	28	T@MIN	Minus
44	2C	T@LGE	Unconditional branch
48	30	T@GT	Greater than
52	34	T@GTE	Greater than or equal
56	38	T@LT	Less than
60	3C	T@LTE	Less than or equal
64	40	T@EQ	Equal
68	44	T@NE	Not equal
72	48	T@OR	Logical OR
76	4C	T@AND	Logical AND
80	50	T@MPY	Multiply
84	54	T@PLS	Add
88	58	T@LEFT	Subscript (left parenthesis)
92	5C	T@TEST	Test compare

Value	Hex	Symbolic Name	Meaning
96	60	T@DADD	Define address
100	64	T@BRA	Branch to source label
104	68	T@COMB	Combine/resolve source label
108	6C	T@BIB	Backward internal branch
112	70	T@FIB	Forward internal branch
116	74	I@RFIB	Resolve forward internal branch
120	78	T@STA	Store address
124	7C	T@ALGN	Align
128	80	T@ENDB	End block
132	84	T@PRLG	Prologue
136	88	T@LIBC	Library call
140	8C	T@SMTB	Symbol table entry
144	90	T@BAL	Branch and Link
148	94	T@PEND	End prologue
152	98	T@EDO	End DO
156	9C	T@BGO	Begin DO
160	A0	T@LOAD	Return
164	A4	T@DVM	Multiply dope vector elements
168	A8	T@DVA	Store into dope vector element
172	AC	T@SPC	Scale complex positive
176	B0	T@SNC	Scale complex negative
180	B4	T@HSA	Halfword subscript
184	B8	T@CDAD	Store code address
188	BC	T@TM	Test under mask
192	C0	T@TITL	Title move (special)
196	C4	T@ORI	OR immediate

Z$LOS If low-order bit is 1, sign of left operand is negative.

Z$ROS If low-order bit is 1, sign of right operand is negative.

Z$TMSK Operand type mask. Has same meaning as dictionary attribute data description field.

Bit 0 If 1, operands are arithmetic.
Bit 1 If 1, operands are variables.
Bit 2 If 1, one of the operands was in error.
Bit 3 If 1, operands are double precision.
Bit 4 If 1, operands are of special type.
Bit 5 If 1, operands are character string.
Bit 6 If 1, operands are floating-point.
Bit 7 If 1, operands are complex.

Z$LOPT Indicates type of value in left operand. The operand types are:

Value	Hex	Symbolic Name	Meaning
0	0	T@NULL	Null
4	4	@ACODE	Code address
8	8	@ASC	Static address
12	C	@AADCN	Adcon address
16	10		Current DSA address
28	1C	T@IDAT	Identifier attribute pointer
40	28	T@PSEU	Pseudo-variable attribute pointer
44	2C	T@TRID	Triad table pointer
48	30	T@DATA	Immediate constant
52	34	T@LIB	Library load table pointer
56	38	T@ERR	Error operand

Value	Hex	Symbolic Name	Meaning
60	3C	T@HADD	Halfword static address
64	40	T@FADD	Fullword static address
68	44	T@BADD	Byte static address
72	48	T@PADD	Immediate DED
76	4C	T@IBIF	In-line built-in function number
80	50	T@CON	Constant table pointer
84	54	T@REG	Register table pointer
88	58	T@ACSDV	Array cross-section dope vector address

Z$LOP Left operand. Contents defined by Z$LOPT.

Z$ROPT Indicates type of value in right operand. The operand types are the same as for Z$LOPT.

Z$ROP Right operand. Contents defined by Z$ROPT.

Z$TSGN If low-order bit is 1, sign of triad is negative.

Z$LUSE The highest triad number which references this triad.

The format of each triad after code is generated is:

```
            1       2       3       4       Byte
        ┌───────────────────────────────────┐
Word 1  │ Z$TOP │ Z$REG │ Z$ROS │ Z$TMSK │
        │───────┼───────────────────────────│
        │       │                           │
     2  │ Z$IBAS│          Z$IOFF           │
        │       │                           │
        │───────────────────────────────────│
        │       │                           │
     3  │ Z$VBAS│          Z$VOFF           │
        │       │                           │
        │───────────────────────────────────│
        │       │                           │
     4  │ Z$VSGN│          Z$LUSE           │
        │       │                           │
        └───────────────────────────────────┘
```

Values for Fields:

Z$TOP Same as before code generation.

Z$REG If nonzero, the register table (R table) entry associated with the register containing the value of the expression. This field is always zero if Z$TOP is the subscript operator.

Z$ROS If nonzero, the register table entry associated with the register containing the value of the offset to the array element referenced by the triad. This field is always zero if Z$TOP is not the subscript operator.

Z$TMSK Same as before code generation.

Z$IBAS Relocation base associated with object code.

Z$IOFF Offset within the code area to the first byte of code which evaluates the expression. This instruction address is not always set for a triad.

Z$VBAS If nonzero, the triad value is in temporary storage and contains the relocation base of that storage. If equal to X'FF', the expression is a subscripted reference.

Z$VOFF Offset within the data area to the value of the triad.

Z$VSGN Sign flags for the value of the triad. If bit 2 is 1, the value is negative in the register. If bit 7 is 1, the value is negative in the temporary storage.

Z$LUSE Same as before code generation.

TITLE: DOPE VECTOR TABLE

Purpose and Usage

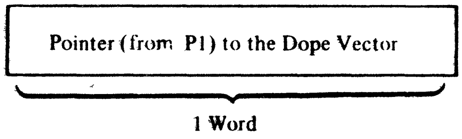
The dope vector table provides the dope vector table pointers to dope vectors of static arrays and strings so that the pointers in the dope vectors may be initialized when static storage is allocated.

Description

The dope vector table is built over the source program in the user's area as the I table and J list are processed during phase 2 of compilation. Each entry in the table contains a pointer to the dope vector (in the constants area) of an array/string in static storage.

Entry Format

The format of an entry in the dope vector table is:



Comments

The last entry in the table is all zeros.

TITLE: ENDFILE TABLE

Purpose and Usage

An ENDFILE table is constructed for each block in which one or more ENDFILE on-units is encountered. Area is reserved for this table in the DSA for the block by the END Generator routine ($EDGN) when the block is being ended. The table is initialized and updated by calls to the IHEONREV routine at runtime.

Description

Each ENDFILE on-unit encountered in a block requires eight bytes in the DSA for the block to build an entry in the ENDFILE table. The number of entries in the table is carried in a pointer to the table.

Entry Format

The format of an entry in the ENDFILE table is shown below.

```
              1       2       3       4       Byte
           ┌───────────────────────────────────┐
    Word 1 │ Action │     A(On-Unit Adcon)     │
           │  Code  │                          │
           │───────────────────────────────────│
           │                                   │
         2 │              A(FCIB)              │
           └───────────────────────────────────┘
```

Values for Fields:

Action Code — If 0, an ON ENDFILE statement referencing the file which corresponds to this entry has not been executed in this block. (See the CALL/360-OS PL/I Language Reference Manual for action that will be performed if the ENDFILE condition is raised.)

If 1, standard system action will be performed if the ENDFILE condition is raised.

If 3, user-specified action will be performed if the ENDFILE condition is raised.

A(On-Unit Adcon) — These bytes are meaningful only for action code 3. Then, they contain a pointer to the on-unit adcon area.

A(FCIB) — This word contains a pointer to the FCIB for the file.

TITLE: ENTRY NAME DECLARATION LIST

Purpose and Usage

An entry name declaration list is established whenever a variable that is a nested entry name declaration is encountered. The list is maintained by the Attribute Node Creation routine ($ANCRE) and allows this routine to be pseudo-recursive. It is necessary because entry name declarations may have entry name declarations within them.

Description

An entry name declaration list is a push-down list. For convenience in storage, this list is kept in the program structure table (P table). The top node of the push-down list is kept in the local variable ANPDL.

Entry Format

The format of an entry name declaration list is shown below.

```
              1         2         3         4     Byte
          ┌───────────────────────────────────────┐
          │                                       │
   Word 1 │                P$SENT                 │
          │                                       │
          ├───────────────────────────────────────┤
          │                                       │
        2 │                P$ZPRM                 │
          │                                       │
          ├───────────────────────────────────────┤
          │                                       │
        3 │                P$ZETK                 │
          │                                       │
          ├───────────────────────────────────────┤
          │         │         │                   │
        4 │  P$ZNP  │         │       P$ZSP       │
          │         │         │                   │
          └───────────────────────────────────────┘
```

Values for Fields:

P$SENT Pointer to the entry attribute node being processed

P$ZPRM Pointer to last parameter attribute node processed

P$ZETK Pointer to current position in entokening of parameter list

P$ZNP Number of parameters previously processed

P$ZSP State of processing (RETURNS attributes or parameter list)

TITLE: ON-UNIT PARAMETER LIST

Purpose and Usage

An on-unit parameter list is constructed for each ON ENDFILE statement encountered in a program. It contains addressing information essential to successful execution of the ON ENDFILE statement.

Description

An on-unit parameter list comprises two words of the static and constants area. It is constructed by the ON Generator ($CON), which stores the address of the list in the block information table (B table). Code is generated to call entry-point IHEONUN of the On-ENDFILE and REVERT Initializer routine (IHEONREV) at runtime, passing to it the location of this on-unit parameter list.

Entry Format

The format of an on-unit parameter list is shown below.

```
              1        2        3        4        Byte
          ┌────────────────────────────────────┐
  Word 1  │ Action │   Pointer to On-Unit      │
          │  Code  │      Adcon Area           │
          │────────┼───────────────────────────│
       2  │ Base   │      Displacement         │
          │ Code   │                           │
          └────────────────────────────────────┘
```

Values for Fields:

Action Code — If 1, standard system action will be performed if the ENDFILE condition is raised.

If 3, user-specified action will be performed if the ENDFILE condition is raised.

Pointer to On-Unit Adcon Area — These bytes are meaningful only for action code 3. Then, they contain a pointer to the on-unit adcon area.

Base Code — If 0C, the remaining three bytes of this word contain a displacement to a location in the adcon area which contains the address of the FCIB for this file.

If 08, the remaining three bytes of this word contain a displacement to a location in the static and constants area which contains the address of the FCIB.

TITLE: ROUTINE ENTRY NAME PROCESSED TABLE

Purpose and Usage

One entry is created in this table for each library runtime routine. The entry corresponds to a fullword entry for the routine in the library load table (L table). The routine entry name processed table is used exclusively by the Runtime Library Loader ($HRTLL) to indicate that a routine is being or has been loaded.

Description

The routine entry name processed table is located immediately following the library load table in the fixed area of working storage. At the start of compilation, this table is set to zeros. It is N bytes in length, where N is the number of entries in the library load table (that is, the number of library runtime routines).

Entry Format

Each entry in the routine entry name processed table is one byte in length. It contains a two-digit hexadecimal action code.

```
      1     Byte
 ┌---------┐
 | Action  |
 |  Code   |
 └---------┘
```

When Runtime Library Loader begins processing an entry in the library load table, it sets the byte corresponding to this entry in the routine entry name processed table to X'01'. Thus, if an entry in the routine entry name processed table is X'01', the routine identified by this entry has been or is being loaded. If an entry contains zeros, no action has been performed on the routine.

APPENDIX C - COMPILER SUPPORT MACROS

The compiler support macros were created solely for the CALL/360-OS PL/I implementers; the CALL/360-OS PL/I user will never come in contact with them.

There are two general categories of compiler support macros: those which are used in handling tables and the others. A general term cannot be given to describe the overall function of the other macros; stated simply, they support implementation of the CALL/360-OS PL/I compiler.

The descriptions in this appendix explain how to use the macros in each category. They do not describe how the macros perform their functions. The macros in each category are described in alphabetic order, according to their mnemonics. The following rules explain the notation that is used.

1. Lowercase letters represent the name of a general class of elements in the CALL/360-OS PL/I language from which a particular entry must be selected by the user.

2. Uppercase letters and punctuation must appear as shown.

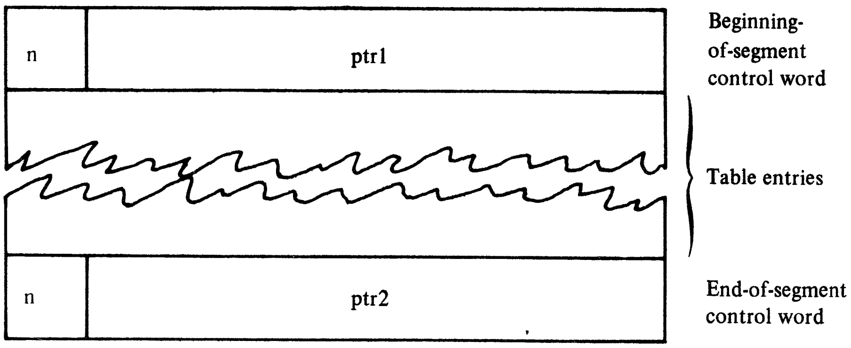
3. Braces { } are used to denote grouping. A vertical stacking of possible entries indicates that a choice is to be made by the user.

4. Square brackets [ ] denote options. Any entries enclosed in brackets may be omitted.

TABLE HANDLING MACROS

EXPANDABLE TABLES

Expandable tables are linearly accessed stacks which are maintained in fixed-size segments. Currently the size of a segment is defined to be 64 words; this size has been chosen to allow the information required for a given table in an average program to be contained in a single segment without overflow. The format of an expandable table segment is:


n
ptr1
Beginning-of-segment control word
Table entries
n
ptr2
End-of-segment control word

The first and last words of the segment are reserved for control purposes. The value of n is the node code for the particular table; ptr1 points to the end-of-segment control word of the preceding table segment and is zero for the first segment in a table; ptr2 points to the beginning-of-segment control word of the following table segment and is zero for the last segment in a table.

Associated with each table is a set of variables. p$HEAD points to the beginning-of-segment control word of the first segment of the table. p$TAIL points to the end-of-segment control word of whichever segment is currently the last segment of the table. p$ACTV points to the first available (unused) word in the current table segment. A two-word variable provides information on the table segment currently accessed; p$CURR contains a pointer to the top of the segment currently being accessed, and p$CURR+4 contains a pointer to the unused space at the bottom of the currently accessed segment. Since the C table and I table contain variable-length entries, the amount of unused space at the bottom of a given segment is not calculable on the basis of entry length. Accordingly, the word immediately preceding the end-of-segment control word is reserved for a pointer to the unused space in these two tables.

The macros to process entries in an expandable table are listed below; each macro is described individually later in this section. Expansion and contraction of table segments is automatically handled by the macros and is transparent to the user.

- GCURR sets a pointer or register to the displacement of the current table entry.

- GNODE uses p$ACTV to acquire space for a new table entry and sets a pointer or register to the displacement of the new entry. Subsequent use of GCURR will obtain a pointer to the new entry. Table segments are automatically expanded, if required. If a new segment is needed, an attempt is made to obtain it from the free pool, that is, list of released segments pointed to by SEGLST of the fixed area. If there are no entries in the list, space is acquired from the area pointed to by FREPTR of the fixed area.

- GPREV uses p$ACTV and p$CURR to step backwards in the table and set a pointer to the entry immediately preceding the entry that was current. Subsequent use of GCURR will obtain a pointer to the new entry. The space occupied by the old entry is released (unless specifically inhibited) and is no longer available. Table segments are automatically returned to the free pool, if required.

- GNEXT uses p$CURR to step forward in the table and set a pointer or register to the new entry. GNEXT can be meaningfully used only after uses of GPREV which did not release the old table entries. Both GPREV and GNEXT signal when they have stepped past the beginning or end of the table.

- MNODE initializes the p$CURR and p$CURR+4 pointers so that GPREV with SAVE option may subsequently be used.

- FAREA frees all segments of the table except the first and re-initializes the first segment for subsequent GNODEs.

- GFRST sets a pointer to the first node of a table and initializes p$CURR and p$CURR+4. GNEXT can subsequently be used to step through the table.

Four compiler support subroutines are used to support the expandable-table macros. $WEXP is used by GNODE to obtain a new segment. $WCTCT is used by GPREV and FAREA to release unused segments to the pool of unused segments. ($WEXP will attempt to obtain a new segment from the pool, if one is available, before it obtains a new space.) $WSTEP is used by GNEXT to step from one segment to the next (to set p$CURR). $WBACK is used by GPREV (with SAVE option) to step from one segment to the preceding segment. (See Section 3, Volume I.)

LISTS

Lists may be single-ended or double-ended. A single-ended list has a universal variable, p$HEAD, associated with it. New entries are added to the head of the list. This kind of list has low overhead in its use. When the list is read, however, entries are returned in reverse order.

A double-ended list has two variables associated with it: p$HEAD and p$TAIL. p$HEAD is set by the user to point to the first item in the list and does not change thereafter. p$TAIL points to the end of the list. New entries are added to the end of the list (with their pointer field set to zero), and p$TAIL is updated with each addition. This list has somewhat higher overhead, but the original order of the entries is maintained.

A set of macros is available to process lists. Maintenance of linkages and head and tail pointers is automatic. To minimize overhead in the macros, the initial setting of p$HEAD in double-ended lists is done by the user.

The macros to process entries in a list are noted below; each macro is described individually later in this subsection.

- GNODE acquires space for a list entry and chains the entry to the head or tail of the list, as specified. (Chaining may be inhibited if desired.) A pointer is set to the new entry.

- GNEXT steps to the next entry in a list which has already been formed, and either sets a pointer to the entry or signals the end of the list.

- DNODE deletes from a list the entry which follows a specified entry. Space used by deleted nodes is not recovered.

- INODE inserts an entry which was not chained by GNODE into a list at the HEAD, TAIL, or following a specified entry. DNODE and INODE can be used jointly to reorder a list or transfer entries from one list to another.

A single-letter symbolic prefix is associated with some fixed tables, all expandable tables, and all lists used by the compiler. The assembler-language labels for all fields within and values associated with a given table or list consistently begin with the single-letter prefix assigned to that table or list. The table-prefix letter codes are as follows:

Code	Meaning
A	Dictionary attribute list
B	Block information table
C	Constant table
D	Line number table
H	Dictionary hash table
I	Initialization table
J	Supplementary initialization list (dope vector list)
L	Library entry name table (library load table)
M	Symbolic instruction table
N	Dictionary name list
O	Operation code table
P	Program structure table
Q	Subscript substitution table
R	Register table
S	Temporary storage table
T	Token table
V	Expression stack
X	Operator stack
Y	Operand stack
Z	Triad table

The tables and lists noted above are discussed in Appendix B of this manual. Other fixed tables, which are also used in CALL/360-OS PL/I, are discussed in Appendix B or in other appendices that deal with closely related topics. These tables and the sections in this manual where they are discussed are noted below.

Table	Explanation in Manual
Attribute Table	Appendix A
Data Parameter Table (also called table of operands or $PTO table)	Appendix A
Dope Vector Table	Appendix B
ENDFILE Table	Appendix B
Entry Name Declaration List	Appendix B
On-Unit Parameter List	Appendix B
Routine Entry Name Processed Table	Appendix B
Symbol Table	Appendix E

TITLE: DELETE ENTRY MACRO (DNODE)

Purpose

The Delete Entry macro is used to delete an entry from a list.

Call

```
[symbol] DNODE  table-prefix

{(register-name) } [, { (register-name) } ]
{pointer-name    }     { pointer-name    }
```

table-prefix:	A, N
register-name:	G2, G3, G4, G5
pointer-name:	any fullword, covered pointer

The second operand points to the list entry which immediately precedes the entry to be deleted. The third operand, if present, will be updated with a pointer to the deleted entry.

Registers Used

G6, G7, P4, P5

Value Returned

Third operand updated to point to deleted entry, if specified.

TITLE: FREE AREA MACRO (FAREA)

Purpose

The Free Area macro releases all space allocated to an expandable table and reinitializes the table in preparation for a subsequent first entry.

Call

```
    [symbol] FAREA  table-prefix

    [,{  (register-name)  }] [,value]
      {  pointer-name     }
```

table-prefix:	B,D,L,P,Q,S,T,V,X,Y,Z
register-name:	G0, G2, G3, G4
pointer-name:	any fullword covered pointer
value:	an absolute expression with a value less than 256

All entries in the specified table are discarded and their space is returned to the free pool. The initial segment of the table is retained and reinitialized for subsequent use.

If a second operand is present, it will be set to a pointer to the first node in the table, and space will be reserved for that node (as in GNODE).

If a third operand is present, the specified value will be placed in the first byte of the first node in the table, and space will be reserved for the node.

Registers Used

G5, G6, G7, P5, C1, C2

Value Returned

Second operand set to point to the first node of the reinitialized table, when specified.

TITLE: CURRENT ENTRY LOCATOR MACRO (GCURR)

Purpose

The Current Entry Locator macro locates the current (most recently constructed) entry in an expandable table.

Call

```
[symbol] GCURR  table-prefix

[,{  (register-name) } ]
  {  pointer-name    }
```

table-prefix:	B,D,I,L,P,Q,S,V,X,Y,Z
register-name:	G0, G2, G3, G4, G5, G6, G7
pointer-name:	any fullword, covered pointer

The second operand is set to point to the current (last constructed) or last unreleased entry in the table. If the second operand is omitted, the output pointer is placed in G7.

Registers Used

G7

Value Returned

Pointer to current table entry, in second operand, if register; in second operand and G7, if pointer; in G7 if second operand is omitted.

TITLE: POINTER TO FIRST NODE MACRO (GFRST)

Purpose

The Pointer to First Node macro establishes a pointer to the first node of a previously constructed expandable table.

Call

```
    [symbol] GFRST  table-prefix

    [,{ (register-name) } ]
      { pointer-name    }
```

table-prefix:	any expandable-table prefix
register-name:	G2, G3, G4, G5, G7
pointer-name:	any fullword covered pointer

The second operand is set to point to the first node of the specified table. If the second operand is omitted, G7 is used.

Registers Used

G6, G7, if second operand is a pointer-name.

Value Returned

Pointer to first node of table, in second operand (and in G7 if second operand is a pointer-name).

TITLE: GET NEXT ENTRY MACRO (GNEXT)

Purpose

The Get Next Entry macro steps to the next entry in a table or list which has already been constructed.

Call

```
    [symbol] GNEXT  table-prefix

       ┌                      ┐     ┌                      ┐
    [, │ (register-name-1)   │ ] [, │ (register-name-2)   │ ]
       │ pointer-name-1      │      │ pointer-name-2      │
       └                      ┘     └                      ┘
```

table-prefix:	A,B,C,D,H,IA,IC,L,N,P,Q,S,V,X,Y,Z
register-names:	G2, G3, G4
pointer-names:	any fullword, covered pointer

Note: References to the I table are either IA or IC. IA is used for a two-word adcon entry; IC is used for all other entries.

For lists, the second operand is assumed to contain a pointer to a given list entry. The macro updates the second operand to point to the immediately succeeding list entry and sets the condition code to nonzero. If the end of the list is reached, the second operand and condition code are set to zero. If the second operand is omitted, the input pointer is assumed to be p$HEAD and the output pointer is placed in G7.

For tables, the second operand is assumed to contain a pointer to a given table entry. The macro updates the pointer to the next table entry and sets the condition code to nonzero. If the end of the table is reached, the second operand and condition code are set to zero. The second operand may not be omitted for tables. The first call to GNEXT should be preceded by calls to either MNODE (followed by GPREV) or GFRST.

Table entries IC and C only, being of variable length, require the length of the current node to be specified by a third operand, either register or pointer. The third operand is not examined for any tables except IC and C.

Registers Used

G5, G6, G7, P5, C1, C2

Values Returned

Pointer to next entry in second operand (G7, if defaulted for lists); condition code set to zero at end of table or list; nonzero, otherwise.

TITLE: GET NODE MACRO (GNODE)

Purpose

The Get Node macro dynamically acquires working storage for the construction of table or list entries, including automatic management of expandable table segments and list linkages.

Call

[symbol] GNODE table-prefix

$$[,\left\{\begin{array}{l}\text{(register-name-1)}\\ \text{length}\end{array}\right\}]\ [,\left\{\begin{array}{c}\text{HEAD}\\ \text{TAIL}\\ \text{FREE}\end{array}\right\}]\ [,\left\{\begin{array}{l}\text{(register-name-2)}\\ \text{pointer-name}\end{array}\right\}]$$

table-prefix:	A,B,C,D,H,IA,IC,J,L,N4,N8,P,Q,S,T,V,X,Y,Z
register-name-1:	G2, G3, G4
length:	an absolute expression
register-name-2:	G2, G3, G4, G7 (should not duplicate register-name-1)
pointer name:	any fullword, covered pointer

Note: References to the I table are either IA or IC. IA is used for a two-word adcon entry; IC is used for all other entries.

When the second operand is omitted, the standard length for each entry is used, except for A, IC, C, and J, for which a length must be specified.

Use of register-name-1 implies that the length of the table entry is contained in the specified register. An explicit length operand overrides the standard length for all tables.

For the token table (T table) only, each call to GNODE acquires a new table segment and returns a pointer to the first usable word within the new segment.

For lists, the node obtained is chained to whichever end of the list is specified by the third operand. The operand FREE inhibits chaining. If the third operand is omitted, HEAD is assumed. The third operand is ignored for tables.

The fourth operand optionally specifies where the value returned by the macro will be placed. If the fourth operand is omitted, register G7 is used.

Registers Used

G5, G6, G7, P5, C1, C2

Value Returned

Pointer to the first word of the new entry in the fourth operand, if the fourth operand is a register; in G7 and the fourth operand, if the fourth operand is a pointer; and in G7 if the fourth operand is omitted.

TITLE: GET PREVIOUS ENTRY MACRO (GPREV)

Purpose

The Get Previous Entry macro obtains the previous entry in a table.

Call

```
    [symbol] GPREV  table-prefix

       ⎧ (register-name)              ⎫
       ⎪ pointer-name                 ⎪
    [, ⎨ ⎧ (register-name)⎫           ⎬  ] [,label]
       ⎩ ⎩ pointer-name   ⎭ ,SAVE     ⎭
```

table-prefix:	B,P,S,X,Y,Z,Q,V,I,L,D
register-name:	G2, G3, G4
pointer-name:	any fullword, covered pointer

When the SAVE operand is not specified, the entry in the table which was current when the macro was called is released to the free storage pool, and its contents are not subsequently available. The preceding entry is made the current entry.

If the second operand is omitted, the output pointer is placed in G7; if a register or pointer name is given, the output pointer is placed in the second operand.

If the SAVE operand is specified, storage is not released, and table entries are subsequently available for later use. The register or pointer named in the second operand must point to a desired table entry; the macro updates the pointer or register to the table entry preceding the one originally pointed to. Entries stepped over may later be recovered using the GNEXT macro. If SAVE is specified, a pointer or register must also be specified, and the pointer or register must have been initialized prior to the first GPREV call by a call for MNODE to mark the entry at which reversal begins.

If a label is specified as the fourth operand, control will be transferred to that label when an attempt is made to back off the beginning of the table.

Registers Used

G5, G6, G7, P5, C1, C2

Values Returned

Pointer to table entry in second operand, if register; in G7 and the second operand, if pointer; in G7 if the second operand is omitted (when not using SAVE option). If GPREV backs off the beginning of the table, the second operand and the condition code are set to zero. Otherwise the condition code is set to nonzero.

TITLE: INSERT ENTRY MACRO (INODE)

Purpose

The Insert Entry macro inserts an entry into a list.

Call

```
    [symbol] INODE  table-prefix,

    ⎧ (register-name) ⎫    ⎧ (register-name) ⎫
    ⎩ pointer-name    ⎭[,  ⎨ pointer-name    ⎬ ]
                           ⎪ HEAD            ⎪
                           ⎩ TAIL            ⎭
```

table-prefix:	A,N
register-name:	G2, G3, G4
pointer-name:	any fullword, covered pointer

The second operand points to the entry to be inserted. The third operand points to the list entry <u>following which</u> the new entry is to be inserted. The third operand may also simply specify the HEAD or TAIL of the list. If the third operand is omitted, TAIL is assumed.

If HEAD or TAIL is specified, p$HEAD or p$TAIL will be updated by the macro. Otherwise the operands remain unchanged.

Registers Used

G5, G6, G7, P4, P5

Value Returned

Updated p$HEAD or p$TAIL, if the third operand so specified. Otherwise, none.

TITLE: ESTABLISH POINTER MACRO (MNODE)

Purpose

The Establish Pointer macro has two purposes:

1. To establish pointers for subsequent use of the GPREV macro with SAVE option.
2. To reestablish pointers in p$CURR and p$CURR+4 which may have been altered. GPREV with the SAVE option causes alteration of the p$CURR pointer when a segment boundary is crossed. Thus GPREV with SAVE option followed by GPREV without SAVE must have an intervening MNODE macro to reestablish pointers to the segment in effect before GPREV with SAVE was issued.

Note: Successive uses of the GPREV macro with SAVE option must not be separated by an intervening MNODE macro.

Call

```
[symbol] MNODE  table-prefix,

 { (register-name) }
 { pointer-name    }
```

table-prefix:	B,P,S,X,Y,Z,Q,V,I,L,D
register-name:	G2, G3, G4
pointer-name:	any fullword, covered pointer

MNODE places a pointer to the current node in the second operand and establishes internal controls so that subsequent GPREV macros with the SAVE option can step backwards in the table non-destructively.

Registers Used

G6, G7 (Second operand may specify G7.)

Value Returned

None

OTHER MACROS

Compiler support macros that perform functions other than table handling are described below.

TITLE: SUBROUTINE CALL MACRO (CALL)

Purpose

The Subroutine Call macro provides linkage between subroutines in the CALL/360-OS PL/I compiler.

Call

```
    [symbol] CALL

    { adcon-name  }
    { $adcon-name }
    { @adcon-name }
```

adcon-name: name assigned to the adcon for the entry point of the desired subroutine

By convention, the names of all entry-point adcons in the compiler begin with @. Entry-point names themselves begin with $. The Subroutine Call macro will accept a name with or without an @, or with a $, and convert it to the proper form.

The coding used in the calling sequence is as follows:

```
    L       C1,adcon
    BALR    C1,C1
```

Registers Used

C1

Value Returned

None

TITLE: SVC INTERFACE MACRO (CSVC)

Purpose

The SVC Interface macro provides a uniform interface for SVC invocation for compiler routines, regardless of operating environment.

Call

```
    [symbol] CSVC  svc-code-number

    svc-code-number:  operand number of the desired SVC
```

The macro generates a call upon an SVC interpreter subroutine. In a simulated environment, the SVC is also simulated. In the real environment, a live SVC is given. By using an interface, the necessity for two versions of compiler routines is avoided.

Registers Used

None

Value Returned

None

TITLE: DED MACRO (DED)

Purpose

The DED macro changes a compiler data descriptor to a DED for the library.

Call

[symbol] DED

$$\left\{\begin{array}{l}\text{(register-name)}\\ \text{pointer-name}\end{array}\right\}$$

register-name: any register (Only the high byte (bits 0-7) changed.)

pointer-name: any covered byte pointer

The byte indicated or the high byte of the register indicated is changed from a compiler data descriptor to a DED acceptable to the library.

Registers Used

None

Value Returned

None

TITLE: EXPRESSION PROCESSOR CALL MACRO (EXPG)

Purpose

The Expression Processor Call macro generates a call to the Expression Processor Controller ($NEXP).

Call

    [symbol] EXPG expression-type, result-type, label

expression-type:	code for the type of expression:	
	V@EXP	expression
	V@ASS	assignment
	V@AASS	array-assignment
result-type:	data descriptor byte for the type of expression desired	
label:	covered label	

A call is generated to the Expression Processor Controller. If the expression is an array expression, return is to the third operand. This macro establishes all information needed by the Expression Processor Controller except the contents of $PTR.

Registers Used

    G0, G5, G6, G7, P5, C1, C2

Value Returned

None

TITLE: FORWARD INTERNAL BRANCH MACRO (FIB)

Purpose

The Forward Internal Branch macro creates a forward internal branch triad.

Call

```
    [symbol] FIB

    { (register-name) }
    { pointer-name    }  , branch-code
```

register-name: G2, G3, G4 (Contains an offset from P1.)

pointer-name: any fullword pointer in fixed working storage

branch-code: value for branch code (Use as right-operand of GTRD macro.)

The contents of the location indicated by the first operand are placed in the left operand of the FIB triad. The location is filled with a triad pointer pointing to the FIB triad. If the previous contents of the second operand was a triad pointer, the indicated triad is changed so that its last reference word also points to the new triad. This macro calls the Get Next Triad Entry routine ($GTRIAD) to obtain the next available space in the triad table (see Section 3, Volume I).

The second operand is passed intact as an operand to the GTRD macro.

Registers Used

G5, G6, G7, P5, C1, C2

Value Returned

None

TITLE: ERROR INTERFACE MACRO (GENER)

Purpose

The Error Interface macro provides an interface for error messages.

Call

[symbol] GENER	message-number, pointer [, (parameter-list)]
message-number:	number assigned to the error message which is to be printed
pointer:	pointer to an entry in the token table indicating the token at which the error was detected (register notation or a name may be used)
parameter-list:	one, two, or three operands. The operands must be pointers to tokens, name list entries, or attribute nodes (either register notation or named) or a character string of not more than eight characters enclosed in quotes. Only one string may be specified in the parameter list.

The macro prepares the interface with the Error Message Editor ($XERR) by storing the specified pointers and/or string into the error communication area, $ERROR. The pointer is used to obtain the line and column number of the statement in error. The parameter pointers are used to insert variable data into the texts of the error messages.

Registers Used

None

Value Returned

None

TITLE: GET TOKEN MACRO (GETKN)

Purpose

The Get Token macro updates a pointer to the next token in the token table.

Call

[symbol] GETKN

$$\left\{\begin{array}{l}\text{(register-name)}\\ \text{pointer-name}\end{array}\right\}$$

register-name: any G-register except G0

pointer-name: any covered, fullword pointer

The macro will update the first operand to point to the token following the one originally pointed to by the first operand. Line-number tokens are ignored.

Registers Used

P5 (If pointer-name is used, G7.)

Value Returned

Updated value in the first operand, pointing to the token following the input token.

TITLE: GENERATE TRIAD MACRO (GTRD)

Purpose

The Generate Triad macro constructs portions of a triad, as specified.

Call

[symbol] GTRD	operator, left-operand [, right-operand]
operator:	absolute expression less than 256, or a parenthesized register-name (G0, G2, G3, or G4)
left-operand:	a self-defining-term, the name of a field, or a parenthesized register-name (G0, G2, G3, or G4)
right-operand:	same as for left-operand

A call is made to the $GTRIAD routine to obtain the next available space in the triad table. (See Section 3, Volume I.)

The macro fills in the operator byte and the left- and right-operand words of the triad. Operand words must be preformatted (that is, contain the type byte if required) before the macro is called.

Registers Used

G5, G6, G7, P5, C1, C2

Values Returned

Pointer to the constructed triad in G7. Address of constructed triad in P5.

TITLE: SYMBOLIC INSTRUCTION TABLE MACRO (INST)

Purpose

The Symbolic Instruction Table macro allows a convenient notation for the various components of an entry in the symbolic instruction table (M table).

Call

```
[symbol] INST  operation,operand-1,operand-2[,operand-3,
               operand-4]
```

operation: name of an instruction in the operation code table (O table)

operand: a macro-argument sublist of the form:

```
(type,value)
```

where type is a character string which, when prefixed with the characters M@, forms a symbol defined as an absolute value; and value is a self-defining term, or a symbol having an absolute value.

Operand-1 represents the R1 field of the generated instruction; operand-2, the R2 or storage address field; operand-3, if present, the X1 field; and operand-4, the B1 field.

If the operation is XFR, only operand-1 is specified; it must be the name of another instruction in the symbolic instruction table.

If a symbol is specified in the name field, it is defined as the displacement between its location and the base of the symbolic instruction table.

Registers Used

None

Value Returned

None

TITLE: ADCON GENERATION MACRO (RCON)

Purpose

The Adcon Generation macro generates adcons required for the compiler.

Call

```
    string-1   RCON    &Z,string-2
```

string-1: a set of characters which, when prefixed with @, forms the name of the adcon to be used for calling a routine.

string-2: a set of characters which, when prefixed with $, forms the name of a routine's entry point.

$Z: the concatenation parameter defined by the SYMDEF macro (described later in this subsection).

RCON is called by the SYMDEF macro. It generates adcons for compiler routine entry-points. Depending upon the circumstances, these adcons must be either relative to the base of phase 1, the base of phase 2, or, for use in DSECT's, should merely be DS reservations.

If the SYMDEF symbol concatenation parameter (&Z) is not null and is not the letter W, the resulting code is for use in the Phase 1 Initializer ($CCONT). RCON generates adcons relative to phase 1, in the form:

```
    A($entry-$CCONT)
```

If the SYMDEF concatenation parameter is the character W, the resulting code is for use in the Phase 2 Initializer ($WCONT). RCON generates adcons relative to phase 2, in the form:

```
    A($entry-$WCONT)
```

If neither of these conditions is true, the resulting code is for use in DSECT's, and RCON generates a DS statement for a fullword instead of an adcon.

Registers Used

None

Value Returned

None

TITLE: RESOLVE FORWARD INTERNAL BRANCH TRIAD MACRO (RFIB)

Purpose

The Resolve Forward Internal Branch Triad macro creates a resolve forward internal branch triad.

Call

```
    [symbol] RFIB  pointer-name

    pointer-name:  either the name of a fullword covered pointer or
                   the location of a pointer, expressed as 0(G-reg,
                   P-reg)
```

The contents of the pointer-name is placed in an RFIB triad. If the contents of the pointer is a triad pointer, then the last usage word of the indicated triad is changed to point to the RFIB triad. This macro calls the $GTRIAD routine to obtain the next available space in the triad table (see Section 3, Volume I).

Registers Used

G5, G6, G7, P5, C1, C2

Value Returned

None

TITLE: SKIP TOKEN MACRO (SKPTK)

Purpose

The Skip Token macro updates a pointer to the next token of the type specified (at the same parenthesis level), or to the next semicolon, whichever occurs first.

Call

```
[symbol] SKPTK

{ (register-name) }
{ pointer-name    }  [,(code-name-1,...,code-name-n)]
```

register-name: any G-register except G0

pointer-name: any covered, fullword pointer

code-name: one of the following:

AND
ASGN
CNST
COL
COM
DELM
DIV
EQ
EXP
GT
GTE
ID
KEY
LPR
LT
LTE
MIN
MPY
NE
NULL
OR
PLS
RPR
SMC

If the second operand is omitted, the macro searches for a semicolon. If only one code-name is specified, parentheses may be omitted from the second operand.

Registers Used

P5; if pointer-name is used, G7

Value Returned

Updated value in the first operand, pointing to a token which is either one of the types specified by the second operand (at the same parenthesis level), or a semicolon.

TITLE: SYMBOL DEFINITION MACRO (SYMDEF)

Purpose

The Symbol Definition macro defines field and register names, user's area, and compiler working storage and provides USING statements as appropriate.

Call

```
    SYMDEF     (table-prefix-1,...,table-prefix-n),char-1,char-2

    table-prefix:  A,B,C,D,H,I,J,L,M,N,O,P,Q,R,S,T,V,X,Y,Z
```

The presence of a given table-prefix in the first-operand list causes the inclusion of the symbol definitions for that table or list in the current assembly.

char-1: For the benefit of the Controller ($CNT), a non-null character other than W in the second operand causes the compiler's working storage (normally a DSECT) to be replicated as a CSECT (to facilitate initialization) with all symbols prefixed by the character given as the second operand. When char-1 is null, symbol definitions for compiler working storage are brought in as a DSECT, and USING statements are given for P2 covering W$STRT and P0 covering W$STRT + 4092. If char-1 is W, only those adcons required to support the second phase overlay portion of the compiler are assembled, relative to a base in the Phase 2 Initializer ($WCONT). (See Section 3, Volume I.)

char-2: Any non-null value in the third operand causes the inclusion of the symbol definitions in the $$USER area and $$UTT area, and a USING statement for P1 covering $$USERS.

Registers Used

None

Value Returned

None

TITLE: TALLY MACRO (TALLY)

Purpose

The Tally macro tallys a counter, increases by a given amount, and performs boundary alignment when requested.

Call

```
[symbol] TALLY     counter-name

                                        ⎧ 1                  ⎫
    ⎧ value               ⎫             ⎪ 2                  ⎪
[,  ⎨ (register-name-1)   ⎬ ]   [,      ⎨ 4                  ⎬ ]
    ⎩ field-name          ⎭             ⎪ 8                  ⎪
                                        ⎩ (register-name-2)  ⎭

    ⎧ (register-name-3)   ⎫
[,  ⎨                     ⎬ ]
    ⎩ pointer-name        ⎭
```

counter-name: name of the counter (fullword) to be updated

value: a self-defining-term representing the amount by which the counter is to be increased

register-name-1: name of a G-register containing the amount by which the counter is to be increased (G0, G2, G3, G4, G5, G6)

field-name: name of a field (fullword) containing the amount by which the counter is to be increased.

register-name-2: name of a register containing a value of 1, 2, 4, or 8. These values represent the alignment desired. The counter will be increased to the next multiple of the alignment code, if required, before the amount specified in the first operand is added (G0, G2, G3, G4, G5, G6).

register-name-3: any G-register except G7; G7 may be used if the second operand is omitted. The value of the counter after alignment but before the amount is added will be placed in the fourth operand.

pointer-name: any fullword pointer. The value of the counter after alignment but before the amount is added will be placed in the fourth operand.

If the second operand is omitted, the counter is aligned and updated but not increased. If the third operand is omitted, the counter is increased but not aligned. If the second and third operands are both omitted, the counter is placed without change in the fourth operand (which itself defaults under these conditions to G7).

Registers Used

None

Value Returned

None

TITLE: ENTOKENING AND GENER INTERFACE MACRO (TGENER)

Purpose

The Entokening and GENER Interface macro provides an interface between the Entoken routine ($ATKN) and the GENER macro.

Call

```
    [symbol] TGENER    message-number,pointer,parameter-list
```

The parameters are those required for the Error Interface macro (GENER). (See "Error Interface Macro (GENER)", above.)

During entokening, register G4 contains a pointer relative to a given token table segment. For the purposes of the GENER macro, this pointer must be relative to the base of the user's data area. The TGENER macro adjusts register G4 as required, calls the GENER macro, and then restores register G4 to its original condition.

Registers Used

None

Value Returned

None

APPENDIX D - RUNTIME SUPPORT MACROS

All routines of the runtime support library conform to the standards described in the following paragraphs.

GENERAL

The CALL/360-OS PL/I compiler library was developed using the OS/360 F-Compiler Library as a starting point (first version, Level-0). Changes were made to the OS/360 F-Compiler Library design to satisfy different requirements imposed by a different system (the time-sharing system) or when significant improvement in execution performance could be made.

The time-sharing considerations evolved around the following:

- Break-up of the F-level work spaces and code modules into two distinct parts: a) a relocatable part including only address information, and b) a non-relocatable part including the body of the code and all other data and information.
- Reassignment of general fixed-point registers in harmony with the relocatable and non-relocatable classification of data.
- Removal of V-type address references embedded in the code.
- Removal of machine commands in which a general register is set by the command itself, that is, the Edit and Mark instruction, etc.
- Elimination and/or insertion of additional code and data as required.
- Alteration of calling sequences as required.
- Conversion of F-level fixed-point logic to CALL/360-OS fixed-point specifications.
- Renaming of global symbols.
- Formatting of library modules to CALL/360-OS specifications.

The modules that constitute the library provide two basic functions:

1. Interface Services. These modules serve as an interface between compiled code and the facilities of the supervisor. They are described in Volume II under "Library Interface Services" in the section entitled "Runtime Support Summary".
2. Computational Services. These modules perform computational operations on data and shape it to the user's requirements. They are described in Volume II under "Library Computational Services."

The library is designed in a highly modular fashion. Modularity is in terms of functions which can be meaningfully separated and are contained within separate library modules.

NAMING CONVENTIONS

Module (routine) names are composed of a unique combination of three characters that give a mnemonic identification of a module's function. The module names are never employed within any system process; however, they are vital for documentation reference.

Entry names are four characters in length, the first three being those of the module name, and the fourth identifying a specific entry point to the module. All linkages to a library module must reference a specific four-character entry-point name.

For purposes of identification, library module and entry names begin with the letter-set prefix "IHE."

Note: The CALL/360-OS PL/I library routines follow the naming conventions stated above and applied in this manual. However, there are some exceptions in the member names assigned to certain routines when stored in CALL/360-OS PL/I system libraries. The member names are:

IOB, IOD, IOP, IOX, and LDO (referred to in documentation as IHEIOB, IHEIOD, IHEIOP, IHEIOX, and IHELDO).

STORAGE REQUIREMENTS AND LIBRARY ADDRESS CONSTANTS

Library routines require working storage, for the following reasons:

1. During explicit communication between modules, the calling module must provide a non-relocatable storage area commonly called the static storage area (SSA) for the called module to use.

2. Intermediate results must be stored.

3. During implicit communication between modules, there must be a storage area containing common symbols.

The library work space fulfills these functions. It is allocated by the compiler in the user's work area and subdivided into unique storage areas, each of which is pointed to by an address constant in a fixed location in the address constant area.

The work space is divided into two major areas as follows:

1. Relocatable working area (LWSP) (contains relocatable information that must be updated at every relocation of the program in core storage)

2. Non-relocatable work area (LWS) (contains non-relocatable data)

Each of the two major work areas is further allocated by the compiler in the user's work areas and subdivided into unique storage areas, each of which is pointed to by an address constant in a fixed location in the adcon area. Comparable subdivisions of the major areas are paired so that for a given activity both relocatable and non-relocatable space will be available for storing of addresses and other data. Non-relocatable subareas will be identified by four-letter codes and corresponding relocatable areas by the same four-letter codes with the letter 'P' appended.

The library communications area is one of the unique area pairs contained in the LWS.

The library as a whole is highly structured. Each module in the library has an associated level number that strictly determines which unique pair of work spaces the module may use.

Level numbers are assigned by the following rules:

1. A module that calls no other module is assigned Level 0.

2. A module that calls other modules is assigned a level number one greater than the maximum level number of all the modules it calls.

3. A module that calls another module but does not expect a return is assigned the level number of the called module.

There are five unique area pairs in the library work space (LW0/LW0P, LW1/LW1P, LW2/LW2P, LW3/LW3P, and LW4/LW4P) which are used by library modules for an SSA and an intermediate storage area. Modules assigned Level Number 0 may use only LW0 and LW0P, modules assigned Level Number 1 may use only LW1 and LW1P, etc. In this way, it is assured that a library module's SSA will not be destroyed during explicit communication, if the caller expects a return.

Calling the execution error package (EXEP) is not considered sufficient to raise the level number of a library module, because EXEP has unique storage areas of its own (called LWE and LWEP).

Figure D-1 specifies the address constants in the address constants area, which points to the base addresses of unique areas of the library work space, and the functions of those areas.

Address Constant	Function of Unique Area Pointed to
IHEQLWS IHEQLWSP	Pointer to major library work spaces
IHEQLCA IHEQLCAP	Library communications areas (also known as library common areas)
IHEQLWE IHEQLWEP	SSA and working storage for execution error package (EXEP)
IHEQLSA IHEQLSAP	Reserved space not used at present
IHEQLW0 IHEQLW0P	SSA and working storage for Level Number 0 library modules
IHEQLW1 IHEQLW1P	SSA and working storage for Level Number 1 library modules
IHEQLW2 IHEQLW2P	SSA and working storage for Level Number 2 library modules
IHEQLW3 IHEQLW3P	SSA and working storage for Level Number 3 library modules
IHEQLW4 IHEQLW4P	SSA and working storage for Level Number 4 library modules

Figure D-1. CALL/360-OS PL/I Address Constants Area

(See "The Library Work Space", below, for a complete description of the library work space.)

DATA REPRESENTATION

By virtue of declared attributes, data may exist in the following forms within a CALL/360-OS PL/I program:

1. Arithmetic data

   a. Real fixed
   b. Real float
   c. Complex fixed
   d. Complex float

2. Character-string data

3. Statement-label data

The following representations are available internally to the IBM System/360:

1. Floating-point (long and short)

2. Binary fixed-point

3. Packed decimal

4. Character string

The relationships between the forms declared for a data item in the CALL/360-OS PL/I program and the actual representation used internally to the IBM System/360 are shown in Figure D-2.

CALL/360-OS PL/I Compiler Forms	IBM System/360 Forms				
	Binary Fixed-Point (4 bytes)	Packed Decimal (9 bytes— 16 digits plus sign)	Short Floating-Point (4 bytes)	Long Floating-Point (8 bytes)	Character String (Maximum 256 bytes)
CALL/360-OS PL/I Forms:					
Internal:					
Arithmetic:					
Fixed-Point	X				
Floating-Point			X	X	
String:					
Character					X
External:					
Arithmetic:					
F-Format					X
E-Format					X
String:					
A-Format					X
Special Library Intermediate Forms:					
Binary Intermediate				X	
Decimal Intermediate		X			

Figure D-2. CALL/360-OS PL/I Data Representation

Library support macros are concerned with the following functions:

1. Exchange information between the phase 2 compilation wrap-up and runtime library modules.

2. Facilitate loading of library working storage covers.

3. Facilitate branching within the library.

4. Define DSECT's and constants universally applicable within the library.

5. Ensure uniformity in the performance of certain special functions, that is, calculate the difference between two addresses, etc.

(See "Library Support Macros", below, for complete description of all library support macros.)

THE LIBRARY WORK SPACE

RELOCATABLE WORK AREA (LWSP)

The DSECT name is IHELIBP and is described by the following table. (The variable names are ordered within the DSECT as they appear in the table.)

DSECT Variable Name	Hex Off-Set	Area Size (Bytes)	Explanation
WBR1	0	4	Second transfer vector. (Used by the arithmetic conversion package (ACP).)
WBR2	4	4	Third transfer vector. (Used by the ACP.)
WRCD	8	8	A(Target), A(Target DED). (Used by the ACP.)
WFDT	10	4	A(Target FED). Implicit parameter for F- or E-format output conversion. (Set by F/E-format and string directors for use by ACP.)
WFED	14	4	A(Source FED). Implicit parameter for F- or E-format input conversion. (Set by F/E-format and string directors for use by ACP.)
WFCB	18	4	A(File Control Block). (Used by C-format, F/E-format, and string input/output directors.)
WCNP	1C	4	A(First and Last Address Pair). (Set by the I/O directors.)
WCN1	20	8	A(Start of Real Part of a C-Format Data Item), A(End of Real Part of a C-Format Data Item). (Used by the C-format directors and the F/E-format input director.)
WCN2	28	8	A(Start of Imaginary Part of a Complex Data Item), A(End of Imaginary Part of a Complex Data Item). (Used by the C-format directors and F/E-format input director.)
WTEMP	38	8	Erasable storage. Not used across calls.
WJXIDVA	40	4	A(Array Dope Vector). (Used by the interleaved array indexing routine.)
WJXILADD	44	4	A(Last Array Element Returned). (Used by the interleaved array indexing routine.)
ZLWEP	48	80	Relocatable work space for the execution error package. (Shared by the EXEP and the C-format directors.)
ZLSAP	98	80	Level-3 and level-4 routines scratch space.
ZLW0P	E8	80	Temporary address storage for library level-0 modules.
ZLW1P	138	80	Temporary address storage for library level-1 modules.
ZLW2P	188	80	Temporary address storage for library level-2 modules.

DSECT Variable Name	Hex Off- Set	Area Size (Bytes)	Explanation
ZLW3P	1D8	80	Temporary address storage for library level-3 modules.
ZLW4P	228	80	Temporary address storage for library level-4 modules.
ZCNTP	278	variable	Relocatable work size. (Used by the compilation wrap-up modules of phase 2 and the load module of the runtime library to reference the end of the relocatable library work space.)

NON-RELOCATABLE WORK AREA (LWS)

The DSECT name is IHEZLIB and is described by the following table. (The variable names are ordered within the DSECT as they appear within the table.)

DSECT Variable Name	Hex Off- Set	Area Size (Bytes)	Explanation
WINT	0	9	Packed decimal intermediate (PDI) or floating-point intermediate (FLI) number storage. (Used by ACP.)
WSCF	C	4	Scale factor associated with the packed decimal intermediate number. (Used by the ACP.)
WSDV	10	8	String Dope Vector (SDV). (Used by the input/output conversion directors.)
WCFD	18	4	Format Element Descriptor (FED). (Used by eight 1-bit intermodular communication switches.) (Bit-7 is the complex switch set by the C-format director to control processing of the complex components by the F/E-format directors. Bit-5 is the update switch set by the string directors to control zeroing of and/or pointing to the various components of the complex item.)
WSWA	1C	1	Eight 1-bit intermodular communication switches.
WSWB	1D	1	Eight 1-bit general purpose switches. (Used by I/O.)
WSWC	1E	1	Eight 1-bit intramodular switches. Not used across calls.
WBUFF	1F	256	Intermediate character storage. (Used by the real output directors.)
WCOUNTI	120	4	Print file current line character count. (Used by output directors.)
WLNEWDTH	124	4	Print file line width. (Used by output directors.)

DSECT Variable Name	Hex Off-Set	Area Size (Bytes)	Explanation
WTERBUFS	128	4	Terminal buffer size. (Used by I/O directors.)
WDISBUFS	12C	4	Disk buffer size. (Used by I/O directors.)
WTOTCHAR	130	4	Current terminal buffer length. (Used by I/O directors.)
WSPEC	134	4	Internal file current field counter.
WCOUNTDK	138	4	Disk file current line length. (Used by output directors.)
WTOTCHDK	13C	4	Current disk buffer length. (Used by I/O directors.)
ZLWE	140	176	Temporary non-address storage for the execution error package.
ZLSA	1F0	80	Not used.
ZLW0	240	176	Temporary non-address storage for library level-0 modules.
ZLW1	2F0	176	Temporary non-address storage for library level-1 modules.
ZLW2	3A0	176	Temporary non-address storage for library level-2 modules.
ZLW3	450	176	Temporary non-address storage for library level-3 modules.
ZLW4	500	176	Temporary non-address storage for library level-4 modules.
ZCNT	5B0	variable	Non-relocatable work space size. (Used by the load modules.)

REGISTERS AND OFFSETS

Assignments are defined in the following chart.

Save Registers

Offset	Mnemonic	Value	Use Definitions
OFP0	P0	6	Cover first page of object code
OFP1	P1	7	(Not specifically assigned)
OFP2	P2	8	(Not specifically assigned)
OFP3	P3	9	Cover address constants
OFP4	P4	10	(Not specifically assigned)
OFP5	P5	11	
OFP6	P6	12	
OFP7	P7	13	Parameter register
OFP8	P8	14	Return register
OFP9	P9	15	Branch register

Fixed Data Registers

OFG0	G0	0	(Not specifically assigned)
OFG1	G1	1	
OFG2	G2	2	
OFG3	G3	3	
OFG4	G4	4	
OFG5	G5	5	

Floating-Point Data Registers

OFF1	F1	0	(Not specifically assigned)
OFF2	F2	2	
OFF3	F3	4	
OFF4	F4	6	

Register P3 must contain cover address constants as indicated at all times. Register P6 is used to load covers required for saving register constants at entry to any library routine; hence, its value is destroyed - not preserved over a call. All other address and data registers are preserved over a library call. Floating-point registers are not saved over a library call.

LIBRARY SUPPORT MACROS

The library support macros were created solely for the CALL/360-OS PL/I implementers; the CALL/360-OS PL/I user will never come in contact with them.

The descriptions given below tell how to use each macro. They do not tell how each macro performs its functions. The macros are described in alphabetic order, according to their mnemonics. The following rules explain the notation that is used.

1. Uppercase letters represent

   a. entries that must appear exactly as shown (for example, CALLERR or IHEBRA).

   b. a general class of entries from which a particular entry must be selected by the user as explained in text which follows the notation. (For example, the parameter OFF shows the place where the offset to the relocatable LWS must be specified in the CALL/360-OS macro call.)

   c. a combination of a and b, where the portion preceding an equal sign is a keyword that must appear as shown and the portion following the equal sign represents a general class of entries from which the user must select a specific entry. (For example, in BR=P9, BR is a keyword parameter and must appear as shown; P9 represents a working register that must be specified.)

2. Braces { } are used to denote grouping. A vertical stacking of possible entries indicates that a choice is to be made by the user.

3. Square brackets [ ] denote options. Any entries enclosed in brackets may be omitted.

TITLE: CALL ERROR MACRO (CALLERR)

Purpose

The Call Error macro develops a call to the execution error package (EXEP) which results in printing an error message and appropriate transfer of control. The user specifies the general data register to be loaded with the error code prior to execution of the Call macro (IHECAL) transferring control to EXEP.

Call

```
    [label]  CALLERR  REG,INDEX,OFFSET,OFFSET1
```

The general data register specified by parameter REG is loaded with the error code set forth in parameter INDEX. The two parameters OFFSET and OFFSET1 serve the same function as the parameters defined for the Call macro. In fact, the Call macro is invoked by the Call Error macro.

Registers Used

One specified general data register

Pseudo registers P3, P6, P8, and P9, as follows:

P3	Adcon error register
P6	Scratch register restored from adcon area
P8	Link register restored from adcon area
P9	Branch register restored from adcon area

TITLE: CALL/360-OS MACRO (CALRTS)

Purpose

The CALL/360-OS macro calls the CALL/360-OS system to request execution of an SVC.

Call

```
[label]  CALRTS  SVC1,OFF,OFF1
```

Parameter SVC1 passes the value of the SVC call.

Parameter OFF is the offset to the relocatable LWS,

Parameter OFF1 is the offset to the non-relocatable LWS.

This macro generates a call to the IHESVC routine.

The macro statement may be labeled.

Registers Used

None

TITLE: CHECK FCB MACRO (CKFCB)

Purpose

The Check FCB macro tests the FCB. If a disk file, a check for empty buffer is made. If empty, an SVC 2 is issued to read a record from disk. The receiving buffer displacement is in register 2 (displacement from communications area to the buffer area). Upon return, byte 13 of the file control interface block (FCIB) will be set with a code as follows:

0 = Read successful
1 = Unrecoverable I/O error
2 = End of data
3 = Read not done because file type is output

This macro updates buffer pointers and returns.

If a terminal file, a question mark is inserted in the output stream and the buffer pointer in the communications area is updated. Then an SVC 2 is requested for input from the terminal unit.

Call

CKFCB (This macro invokes the CALL/360-OS macro (CALRTS).)

Registers Used

Preset registers:

Pseudo registers P3, P6, P7, and P9

Non-preset registers:

Pseudo registers: P2, P4, and G1
Absolute general registers: 2,3

TITLE: ADDRESS CONSTANTS MACRO (IHEADC)

Purpose

The Address Constants macro defines the displacement for each symbol appearing in the address constant area (ADCON) beginning with the alphameric characters "L@". This macro is invoked by the Symbol macro (IHESYM), which defines all of the runtime entry names for which space is to be reserved in ADCON and for which a corresponding "L@" symbol is associated.

Call

```
    IHEADC    LIB1,LIB2,LIB3,LIB4,LIBB1,LIBB2
```

Each parameter field contains one or more subfields, each of which contains an entry-point name. The first three fields are reserved for entry points to modules that are accessible directly through a single reference to ADCON. The last two fields are reserved for entry points to modules requiring an indirect access through an appropriate block adcon area. The ordering of the symbols within the subfields is critical to modules that perform compilation wrap-up and load functions.

Registers Used

None

TITLE: BRANCH MACRO (IHEBRA)

Purpose

The Branch macro saves the contents of a specified general register over a generated Branch and Link instruction and/or assembles the branch instruction using designated registers for branching and linking.

Call

```
    [label]  IHEBRA  LXR,BXR,LOC,P6=P6
```

Parameters LXR and BXR are two general address registers designated as link and branch registers, respectively. Keyword parameter P6 designates a general address register whose contents are to be saved over the branch. Parameter LOC specifies a storage word for saving register content. The default values are:

```
    LXR=P8
    BXR=P9
    P6 =P6
```

Parameter LOC in default results in generation of the Branch and Link instruction only. Caution should be exercised when placing USING statements immediately behind Branch macro statements.

The macro statement may be labeled.

Registers Used

Two or three user-designated general address registers

TITLE: BAA EXTERN MACRO (IHEBXT)

Purpose

The BAA Extern macro loads a specified general address register with the address of an entry point for a module requiring access to a special-function block address constant area.

Call

```
    [label]  IHEBXT  DISP,REG,BR=P9,P3=P3,LOC=#2
```

The first field value must be the relative entry number of the desired entry-point symbol in the external symbol table of the Trailer macro (IHETLR) for the module in which the BAA Extern macro is embedded. The second field is a general data register assigned as a working register. Both fields must be present. Keyword parameter BR specifies a working register, while keyword parameter P3 specifies a general address register containing the cover for ADCON. Keyword parameter LOC points to the beginning of the external symbol table generated by the Trailer macro. Default values are:

```
    BR =P9
    P3 =P3
    LOC=#2 (Default value for keyword parameter NAM2 of the Trailer
            macro)
```

The macro statement may be labeled.

Registers Used

Two specified general address registers
One specified general data register

TITLE: CALL MACRO (IHECAL)

Purpose

The Call macro assembles instructions required to save pseudo register P8 over the branch; loads pseudo register P9 with the branching address; and restores pseudo registers P9 and P6 from specified addresses covered by pseudo register P3 (the covering register in the adcon area).

Call

```
    [label]  IHECAL  VADD,OFFSET,OFFSET1
```

All three parameter fields are displacements with respect to the adcon area cover and must contain valid information prior to execution of the macro.

The macro statement may be labeled.

Registers Used

Pseudo registers P3, P6, P8, and P9, as follows:

P3	Adcon cover register
P6	Scratch register restored from adcon area
P8	Link register restored from adcon area
P9	Branch register restored from adcon area

TITLE: DOUBLE COVER MACRO (IHEDCV)

Purpose

The Double Cover macro loads two adjacent general address registers with the covers for the library non-relocatable and relocatable work spaces, respectively.

Call

```
    [label]  IHEDCV  FIELD,REG
```

Parameter FIELD specifies the library level of the module (that is, LWE, LWS, LW0, LW1, LW2, LW3, or LW4). Parameter REG designates the register to contain the library non-relocatable work space cover. The next higher-numbered register will contain the cover for the library relocatable work space.

The macro statement may be labeled.

Registers Used

Two designated general address registers

TITLE: DIFFERENCE MACRO (IHEDIF)

Purpose

The Difference macro calculates the difference between the contents of two specified address registers, and stores the result into a designated target.

Call

```
    [label]  IHEDIF  R1,R2,NR1,NR2,AREA=WTEMP
```

Parameters R1 and R2 are two general address registers containing the address for which the difference C(R1)-C(R2) is desired. (C(R1) denotes contents of R1, etc.) NR1 and NR2 are two general data registers assigned as working registers. The keyword parameter AREA points to a two-word block of temporary storage. Parameters R1 and NR1 must be specified. Default values for the other parameters are:

```
    R2  =R1+1
    NR2 =NR1+1
    AREA=WTEMP (Doubleword erasable storage in LCA)
```

The macro statement may be labeled.

Registers Used

Two user-specified general address registers
Two user-specified general data registers

TITLE: ERRCD MACRO (IHEERRCD)

Purpose

The ERRCD macro has two functions:

1. Set the error code in LWE.
2. Set the error code in LWE, then branch to Error Routine (IHEERR). (See Section 5, Volume II.)

Call

```
    [label]  IHEERRCD  INDEX,PREG,ROUTNAM,FCIB
```

The first parameter provides the index of the error code. The second parameter provides an address register which is used to cover LWE. The third parameter is the last four characters of the desired entry point name of IHEERR. The fourth parameter is the register containing the FCB. It is changed to point to the FCIB for IHEERRB.

The macro statement may be labeled.

Registers Used

One user-designated general address register

P3	Adcon cover register
P8	Link register
P9	Branch register

TITLE: INITIALIZE FILE CONTROL BLOCK MACRO (IHEFCB)

Purpose

The Initialize File Control Block macro changes P7 from address of FCIB to address of FCB, and moves buffer pointers from saved FCB to common FCB.

Call

```
    [label]  IHEFCB
```

Registers Used

G5	Address of FCB offset
P7	Address of FCIB to address of common FCB
P8	Address of saved FCB
P9	Work

TITLE: SAVE FCB POINTERS MACRO (IHEFCIB)

Purpose

The Save FCB Pointers macro has two purposes:

1. Saves the disk buffer pointers from common FCB area.
2. Saves area for FCB if routine called from compiled code.

Call

```
[label]  IHEFCIB  OFFSET1
```

OFFSET1 is the relocatable library work space level used by this routine.

Registers Used

P3	Adcon cover register
P8	Link address
P7	FCIB address
P5	Work
P6	Work

TITLE: LINK ROUTINE MACRO (IHEFROM)

Purpose

The Link Routine macro determines whether a call to a library routine is from compiled code or from another library routine.

Call

```
    [label]  IHEFROM  EXIT
```

EXIT is the exit address if call is from another library routine.

Registers Used

P3	Adcon cover register
P8	Link register

TITLE: EXTERNAL MACRO (IHEEXT)

Purpose

The External macro loads a designated general address register with the address of an entry point through a single reference to the adcon area.

Call

```
    [label]  IHEEXT  DISP,REGP,REGB
```

The first field must be the character following the "L@" alphamerics of an "L@" symbol defined in the adcon area. The second field is the general address register to be loaded, and the third parameter is the general address register containing the adcon cover. Default values are:

```
    REGP=P9
    REGB=P3
```

The macro statement may be labeled.

Registers Used

Two designated general address registers

TITLE: HEADER MACRO (IHEHDR)

Purpose

The Header macro supplies the wrap-up loader with the following information:

1. The size (in bytes) of the module
2. The number of external references by the module
3. The number of entry points

Call

```
    [label]  IHEHDR  NAM1=#1,NAM2=#2,NAM3=#3,NAM4=#4
```

The keyword parameter NAM1 is a symbol defined by the Header macro, whereas keyword parameters NAM2, NAM3, and NAM4 are symbols defined in the Trailer macro (IHETLR) separating the code section, the external reference section, and the entry-point section. These keyword parameters must agree exactly in value with corresponding keyword parameters of the Trailer macro. The default values are:

```
    NAM1=#1
    NAM2=#2
    NAM3=#3
    NAM4=#4
```

The formulas are:

```
    Module size = NAM2-NAM1-4 (in bytes)
    Number of external references = (NAM3-NAM2)/2
    Number of entry points = (NAM4-NAM3)/4
```

The macro statement may be labeled.

Registers Used

None

TITLE: I/O INTERFACE MACRO (IHEIOD)

Purpose

The I/O Interface macro provides an interface between the I/O conversion directors and the input routine IHEIOG and output routine IHEIOD.

The former interface loads a designated general data register with the length-1 of the string to be input and a general address register with the A(FCB); then it branches to the input routine IHEIOG, after which, and upon return, an appropriate string dope vector is constructed.

The latter interface loads two general address registers with the A(FCB) and A(SDV), respectively. Branching to the output routine IHEIOD then proceeds as described above for input.

Call

```
[label]  IHEIOD  FCB=WFCB,SDV=WSDV,LWSP=LWEP,OP=IN,
                 P9=P9,P8=P8,P5=P5,P0=P0,G1=G1,G0=G0,
                 PTEMP=WTEMP,P7=P7
```

The macro invokes the IHEEXT, IHEBRA, and IHEDIF macros. Keyword pointer parameters point as follows:

FCB	points to A(FCB)
SDV	points to A(SDV)
LWSP	points to A(library relocatable work subarea)
PTEMP	points to A(doubleword erasable area)

Keyword parameter switch OP functions as follows:

OP = IN defines an input file
OP ≠ IN defines an output file

Keyword parameter register assignments are:

```
P8 assigned as linkage register                )
P9 assigned as branch register                 |
P7 assigned as parameter register for A(FCB)   |
P5 assigned as parameter register for A(SDV)   }  general addr registers
P0 preset to A (first page of                  |
   object code cover)                          )
G1 assigned as parameter register             )
   for string length-1 and scratch            } general data registers
G0 scratch                                    )
```

The default values are set as follows:

FCB	=WFCB	Pointer in LCAP
SDV	=WSDV	Pointer in LCA
LWSP	=LWEP	Library work area assigned to EXEP
PTEMP	=WTEMP	Scratch area in LCAP
P8	=P8	Return pointer
P9	=P9	Transfer pointer
P5	=P5	
G1	=G1	
G0	=G0	
P7	=P7	

The macro statement may be labeled.

Registers Used

Five user-designated general address registers
Two user-designated general data registers

TITLE: STANDARD OFFSETS MACRO (IHELBE)

Purpose

The Standard Offsets macro equates standard adcon area, relocatable library work space, and non-relocatable library work space offsets to symbols. It also is used to redefine "L@" symbols to "V" type symbols.

Call

```
    IHELBE
```

Registers Used

None

TITLE: LIBRARY MACRO (IHELIB)

Purpose

The Library macro provides the definitions for the symbols and DSECT's required by a majority of the library routines. The list includes those symbols associated with pseudo registers, standard save-area offsets, and "L@" symbols necessary to program execution, together with the DSECT's which may contain them, and the error codes.

Call

    IHELIB

The Library macro invokes the Symbol macro (IHESYM) to define the "L@" symbols (which in turn invokes the IHEADC macro) and the Library Work Space macro (IHELWS) to define library work area DSECT's, pseudo registers, and error codes.

Registers Used

None

TITLE: LIBRARY WORK SPACE MACRO (IHELWS)

Purpose

The Library Work Space macro defines the library work space DSECT's and associated symbols.

Call

```
    IHELWS
```

The macro is invoked by the Library macro (IHELIB).

Registers Used

None

TITLE: MOPP MACRO (IHEMOPP)

Purpose

The MOPP macro defines the DSECT's describing the block adcon area (BAA) and dynamic storage area (DSA).

Call

```
    IHEMOPP
```

Registers Used

None

TITLE: NAME MACRO (IHENAME)

Purpose

The Name macro generates 80 bytes for absolute patching of compiled code and places the literal constants generated-to-date immediately ahead of the patch area. The patch area is word-aligned and filled with zeros.

Call

```
    IHENAME
```

See also the Patch macro (IHEPCH).

Registers Used

None

TITLE: OPEN TEST MACRO (IHEOPENT)

Purpose

The Open Test macro tests file openings to determine whether they are successful. If not, it determines why a file was not opened and gives an appropriate error message.

Call

[label] IHEOPENT

Registers Used

P7	Address of FCIB

Errors Detected

NOT OPENED (124)
DOES NOT EXIST (126)
LOCKED (127)
IN USE (128)
NOT A DATA FILE (130)

TITLE: PATCH MACRO (IHEPCH)

Purpose

The Patch macro generates space for absolute patching of modules. The space generated is set to character pattern DEAD.

Call

```
    [label]  IHEPCH
```

Registers Used

None

TITLE: RETURN MACRO (IHERET)

Purpose

The Return macro restores the general registers (except symbolic register P6 which is destroyed in the restoring process) and floating-point symbolic registers F3 and F4 from a designated standard save area.

Call

```
    [label]  IHERET  OFFSET1,OFFSET2
```

The two parameter fields are offsets relative to the adcon cover pointing to library work space addresses.

The macro statement may be labeled.

Registers Used

Symbolic registers P3, P8, and P9, as follows:

P3 Adcon cover register
P8 Return linkage register
P9 Scratch register

TITLE: RESTORE MACRO (IHERST)

Purpose

The Restore macro restores all general registers to the values contained in a designated save area according to the standard save-area offsets. (It is assumed that the designated area was preset by the prior execution of a Save macro.) The floating-point registers are not restored.

General registers P0 through P9 are restored from the relocatable standard save area, while general data registers G0 through G5 are restored from the associated non-relocatable counterpart.

Call

```
    [label]  IHERST  FIELD,REG
```

The FIELD parameter value must be the last three alphamerics of a library non-relocatable work subarea. The REG parameter value is a general address register.

The macro statement may be labeled.

Registers Used

One user-designated general address register

TITLE: SAVE MACRO (IHESAV)

Purpose

The Save macro stores all general registers (except a designated general address register which is destroyed in the saving process) into a specified standard save area. The save area specified is normally one of the library work subareas. The general data registers are saved in the non-relocatable part of the work space, and the general address registers in the corresponding relocatable part. The floating-point registers are not saved.

Call

```
    [label]  IHESAV  FIELD,REG
```

The FIELD parameter is a non-relocatable part of a library work space (that is, LW0, LW1, etc.), and the REG parameter is a general address register. Generally pseudo register P6 is selected.

The macro statement may be labeled.

Registers Used

One designated general address register

TITLE: SINGLE COVER MACRO (IHESCV)

Purpose

The Single Cover macro loads a designated general address register with the cover address of a specified library work space. Appropriate USING statements are generated flagging the designated register as a cover register to the compiler.

Call

```
    [label]  IHESCV  FIELD,REG
```

Parameter FIELD specifies the library level of the module (that is, LWE, LWS, LW0, LW1, LW2, LW3, or LW4). Parameter REG designates the register to contain the library non-relocatable work space cover.

The macro statement may be labeled.

Registers Used

One user-designated general address register

TITLE: SDR MACRO (IHESDR)

Purpose

The SDR macro saves the contents of general registers (except symbolic register P6 which is destroyed in the saving process) and the two floating-point symbolic registers, F3 and F4.

Call

```
    [label] IHESDR  D1,D2
```

Parameter D1 is a displacement in the adcon area pointing to the desired relocatable LWS. Parameter D2 is a similar displacement pointing to the desired non-relocatable LWS. (See "Save Macro (IHESAV)".)

The macro statement may be labeled.

Registers Used

Pseudo registers P3 and P6, as follows:

P3 is preset to A (adcon area)
P6 is a scratch register

TITLE: SYMBOL MACRO (IHESYM)

Purpose

The Symbol macro provides the Address Constants macro (IHEADC) with a list of entry-point names for which the Address Constants macro is to reserve space in the adcon area and label said space by concatenating the alphameric symbols "L@" to the left of entry name.

Call

```
        MACRO
        IHESYM
* INDIRECT ADCON ADDRESSING MACRO
* CALLS ADCON MAP MACRO IHEADCN
        IHEADC      (DDJA,DDOA,DDOB,DDOC,DDPD,IOAA,IOAT,IOBA,IOBC,      *
              IODP,IOGA,IOXA,IOXB,IOXC,LDIB,LDIC,LDOB,LDOC,DUMP,ERRA,   *
              ERRB,ERRC,ERNA,SADA,SADB,SADC,SADD,SAFC,DCNA,DIAA,DIAB,   *
              DIAZ,DIAY,DIBA,DIBZ,DIMA,DIMZ,DMAA,DNCA,VPFA,IOPB,       *
              ADMP),                                                   *
              (DOAA,DOAB,DOAZ,DOAY,DOMA,DOMZ,DOBA,DOBB,DOBZ,DOBY,      *
              UPAA,UPAB,VCAA,VCSA,VCSB,VFAA,VFBA,VFCA,VFDA,VFEA,VPAA,  *
              VPBA,VPCA,VPEA,VTBA,ABTO,ABMO,ABGO,DZTO,DZMO,DZGO,MXSO,  *
              MXLO,MXFO,MNSO,MNLO,MNFO,XISI,XILI,XIFI),                *
              (XITI,XIMI,XIGI,XISF,XILF,MZGO,XITF,XIMF,MZTO,MZMO,      *
              PDSO,PDLO,PDFO,PDTO,PDMO,PDGO,SMSO,SMLO,SMFO,SMTO,SMMO,  *
              SMGO,YGSS,YGLS,YGFS,YGTS,YGMS,YGGS,YGSV,YGLV,YGFV,YGTV,  *
              YGMV,YGGV,JXIY,JXIA,CSCO,CSMF,CSS2,DIOA,VSCA,ERRR,GPUT,  *
              VPDA,SVCA,DDIB,ERRZ,SADE,RSET),                   PTR219 *
              (OPEN,CLOS,ONUN,REVT,ERRN,ENDF),                  PTR219 *
              (ATS1,ATL1,ATTO,ATMO,ATS2,ATL2,AHSO,AHLO,AHTO,AHMO,      *
              EFSO,EFLO,EXSO,EXLO,EXTO,EXMO,LNSO,LNLO,LNTO,LNMO,L2SO,  *
              L2LO,LGSO,LGLO,SNSO,SNLO,SNTO,SNMO,CSSO,CSLO,CSTO,CSMO,  *
              SQSO,SQLO,SQTO,SQMO),(TNSO,TNLO,TNTO,TNMO,THSO,THLO,     *
              THTO,THMO,CHSO,CHLO,CHTO,CHMO,SHSO,SHLO,SHTO,SHMO)       *
        MEND
```

The six parameter fields contain the last four characters of all of the entry-point names to the runtime library. The first four fields contain those entry points associated with modules accessible through a single access via the generated points associated with modules accessible through a single access via the generated "L@" symbol. The last two fields contain the entry points associated with modules accessible through a second access to the block adcon area assigned for the function performed by the module.

Caution: The ordering of the above entry-point names is critical with respect to compilation wrap-up and load operations. Names in second, third, and fourth fields may be pushed up into the next previous fields but the overall ordering of names must not be altered. Each field can contain a maximum of fifty-two entry names. The macro is invoked by the Library macro (IHELIB).

Registers Used

None

TITLE: TRAILER MACRO (IHETLR)

Purpose

The Trailer macro supplies the wrap-up loader with the following information:

1. Defines three symbols which divide the module into three parts: the body of code, the external reference section, and the entry-point section, such that the data required in the Header macro (IHEHDR) can be calculated. (See "Runtime Routine Structure" in Appendix A.)

2. Generates a pointer in the external reference table which identifies the external reference and provides a linkage if the external reference points to an entry point associated with one of the special functions requiring access to the block adcon area for the functions (to access the entry-point address).

3. Generates a displacement in the entry point table which identifies the entry point, and a displacement within the code body from which the address of the entry point can be determined.

Call

```
    [label]  IHETLR  EXTRN,ENTRY,NAM1=#1,NAM2=#2,NAM3=#3,
                     NAM4=#4,FMT=OBH
```

The parameter EXTRN contains one subfield per external symbol referenced by the module. Similarly, the ENTRY parameter contains a subfield for each entry point defined in the module. The keyword parameter FMT identifies the form of the subfield entries. IF FMT=OBH, the subfield values are the last four characters of the library external or entry-point name. A slot has been provided for other subfield formats as may be required. At present, FMT≠OBH results in the output of a message.

The keyword parameters NAM1, NAM2, NAM3, and NAM4 are as described for the Header macro and must be identical to respective field definitions for the Header macro.

Default parameter values are:

```
    NAM1=#1
    NAM2=#2
    NAM3=#3
    NAM4=#4
    FMT =OBH
```

Registers Used

None

TITLE: ZAP MACRO (IHEZAP)

Purpose

The Zap macro defines symbols and DSECT's covering FCB's, symbol tables, DED's, the communications area, and user terminal tables.

Call

```
    IHEZAP
```

The macro is invoked by the Library Definition macro (LIBDEF).

Registers Used

None

TITLE: LIBRARY DEFINITION MACRO (LIBDEF)

Purpose

The Library Definition macro provides all the symbol and DSECT definitions as described for the Library macro plus other symbols and DSECT's of special interest to the library interface modules. Among these are:

1. Redefinitions of standard adcon-area offsets
2. Redefinitions of standard save-area offsets
3. Redefinitions of V/type symbols
4. Definition of DSECT's covering FCB's
5. Definition of DSECT's covering DED's
6. Definition of DSECT's covering symbol tables
7. Definition of DSECT's covering communications areas
8. Definition of DSECT's covering user terminal tables

Call

```
    LIBDEF
```

The macro invokes the following macros:

Library macro (IHELIB)
Standard Offsets macro (IHELBE)
Zap macro (IHEZAP)

Registers Used

No registers are used directly by LIBDEF. Its relationships to called routines and required values are shown in Figure D-3.

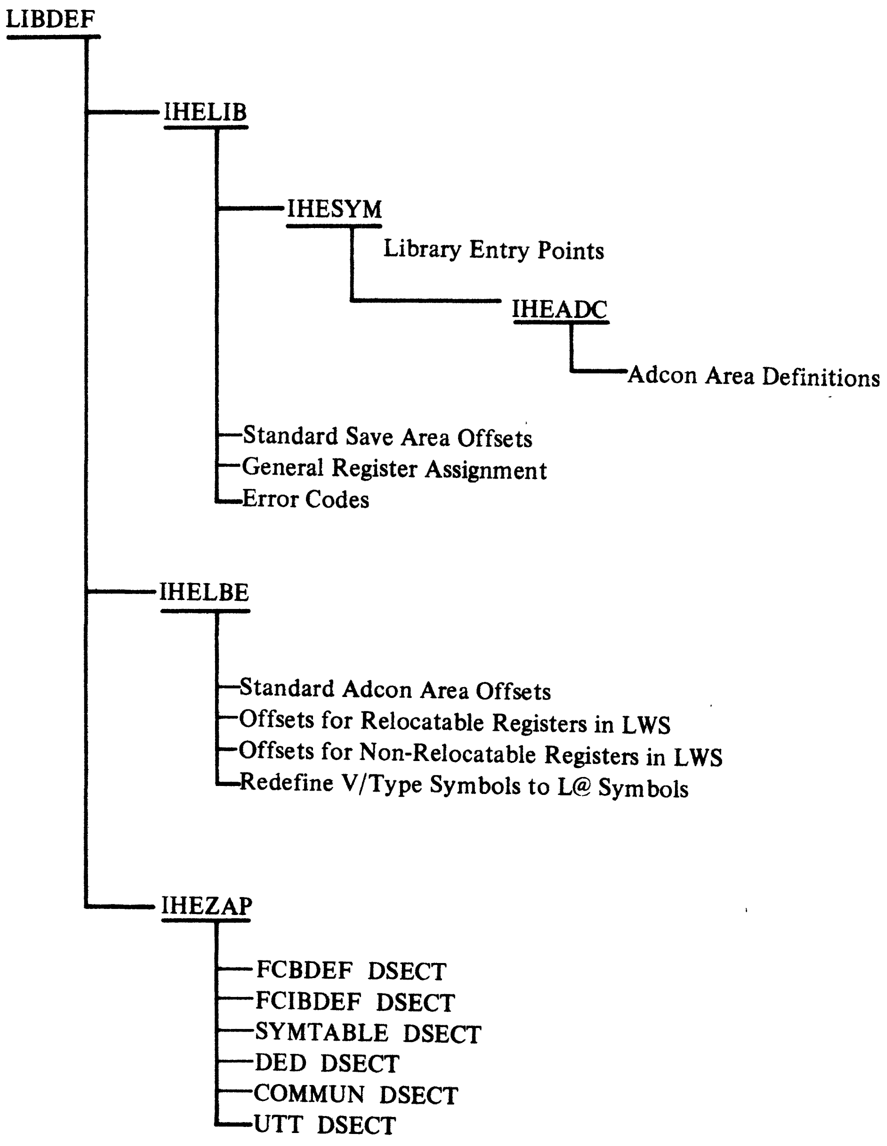

LIBDEF
IHELIB
IHESYM
Library Entry Points
IHEADC
Adcon Area Definitions
Standard Save Area Offsets
General Register Assignment
Error Codes
IHELBE
Standard Adcon Area Offsets
Offsets for Relocatable Registers in LWS
Offsets for Non-Relocatable Registers in LWS
Redefine V/Type Symbols to L@ Symbols
IHEZAP
FCBDEF DSECT
FCIBDEF DSECT
SYMTABLE DSECT
DED DSECT
COMMUN DSECT
UTT DSECT

Figure D-3. LIBDEF Calls

TITLE: READ DISK MACRO (READDISK)

Purpose

The Read Disk macro reads a record from disk.

Call

```
    READDISK  OFF1,OFF2
```

Parameters OFF1 and OFF2 are offsets to the LWS and LWSP area pointing to the non-relocatable and relocatable standard save areas, respectively. The macro invokes the CALL/360-OS macro (CALRTS) and the ERRCD macro (IHERRCD).

Registers Used

Preset pseudo registers:

P3 Adcon cover register
P6 LWS cover register

Scratch pseudo registers:

G2-G5 inclusive

TITLE: READ TERM MACRO (READTERM)

Purpose

The Read Term macro reads a new line from a terminal unit.

Call

```
    [label]  READTERM
```

This macro invokes the CALL/360-OS macro (CALRTS). Prior to invoking the CALL/360-OS macro, a question-mark character is placed in the buffer with all buffer controls updated accordingly.

The macro statement may be labeled.

Registers Used

Preset pseudo registers:

P3 Adcon cover register
P2 A (terminal output control block)
P8 Local cover macro

TITLE: UNIFORM INTERFACE FOR SVC MACRO (RTSSVC)

Purpose

The Uniform Interface for SVC macro provides a uniform interface for SVC invocation from object program routines, regardless of operating environment.

Call

```
[symbol] SVC   svc-code-number

svc-code-number:   the operand number of the desired SVC
```

The macro generates a call upon an SVC interpreter subroutine. In a simulated environment, the SVC is also simulated. In the real environment, a live SVC is issued. By using an interface, the necessity for having two versions of object-program library routines is avoided.

Registers Used

None

TITLE: SET DISK MACRO (SETDISK)

Purpose

The Set Disk macro sets the first eight bytes of the input disk buffer according to information contained in the FCB. If the file is an external disk file, character line counter (WCOUNT1) and total buffer character counter (WTOTCHAR) are reinitialized. A code flag (X'C0') is set in the buffer area.

If the file is an internal disk file, the code flag (X'40') is set in the buffer area and the FCB is updated.

Call

```
    SETDISK
```

Registers Used

Preset pseudo register:

    P3 Adcon cover register

Symbolic registers:

ADLCA	Local cover register
ZERO	General data register

TITLE: SET ERROR CODE MACRO (SETERRCD)

Purpose

The Set Error Code macro sets the error code in the library communications area.

Call

```
    SETERRCD  INDEX,PREG
```

Parameter INDEX supplies the last two characters of the error code, and parameter PREG is a scratch general address which can be used by the macro.

Registers Used

Preset pseudo register P3 as adcon cover register
Symbolic general data register ZERO
A user-designated general address register

TITLE: SET FILE CONTROLS MACRO (SETFLCA)

Purpose

The Set File Controls macro loads two symbolic general data registers (COUNT1,LNEWDTH) with the current line character count and line width, respectively.

Call

SETFLCA

Registers Used

Preset pseudo register:

P3 Adcon cover register

Non-preset:

Pseudo register P4 as local cover register

Symbolic general data registers:

COUNT1	Current line character counter
LNEWDTH	Line width

TITLE: SET DOPE VECTOR MACRO (SETSDV)

Purpose

The Set Dope Vector macro sets the stream dope vector.

Call

```
    [label]  SETSDV  REG1,G1,G2,OFFSET1,OFFSET2,COUNT1
```

Parameter register REG1 supplies the starting address of the sources. Parameter registers G1 and G2 are general-data scratch registers. Parameters OFFSET1 and OFFSET2 supply the offsets in the LWS and LWSP, respectively, pointing to the non-relocatable and relocatable work areas, respectively. Parameter register COUNT1 supplies the source stream length.

The macro statement may be labeled.

Registers Used

Preset pseudo registers:

P0	Adcon cover register
P6	LWS cover register
P9	LWSP cover register

Symbolic general data register COUNT1

One user-designated general address register
Two user-designated general data registers

APPENDIX E - OBJECT CODE STORAGE LAYOUT

This appendix describes the layout of a CALL/360-OS PL/I object program. Each object program consists of distinct sections. These sections, in the order they appear in computer storage, are:

1. Communications area
2. Terminal I/O buffer
3. Object program
4. Line number table
5. Static and constants storage
6. Address constant area
7. CALL/360-OS PL/I library
8. Static array and string storage
9. Disk I/O buffers
10. Dynamic storage

Since the sizes of some sections are not determinable until after compilation has been completed, each section must be addressed separately by different base address constants. The addresses of sections 3, 5, and 6 are always contained in fixed general purpose registers except during execution of a routine from the library. The addresses of all areas are contained in fixed locations in the address constant (adcon) area.

The communications area, terminal I/O buffer, and disk I/O buffers are discussed in Appendix F. All other sections are discussed below.

OBJECT CODE

The object code consists of the machine instructions constituting the compiled program, symbol tables for data I/O, and the object code address vs. line number table for runtime diagnostics.

SYMBOL TABLE

Each symbol table consists of entries for each variable to be written or that can be read in an I/O operation. On input, one table contains entries for all identifiers that can be read. An output operation may use more than one symbol table. Each symbol entry in the symbol table is five words long (fullword-aligned) and contains:

1. The name of the identifier.
2. The DED for the identifier.
3. The number of subscripts.
4. How to locate the identifier, if a scalar, or its dope vector, if an array or a string.

The identifier or its dope vector is obtained by using two offsets. The first offset (K@STO1) indicates the displacement within the adcon area to the base address. The second offset (K@STO2) is the displacement from the base address to the identifier or the dope vector. Figure E-1 shows the format of a symbol table entry.

```
                 1       2       3       4        Byte
         ┌──────────────────────────────────────┐
         |                                      |
  Word 1 |                Name                  |
         |----------- (in EBCDIC)   ------------|
         |              K@STNM                  |
       2 |                                      |
         |--------------------------------------|
         |                        | Number of   |
       3 |          DED           | Subscripts  |
         |         K@STDD         |  K@STSB     |
         |--------------------------------------|
         |             Offset(1)                |
       4 |               K@STO1                 |
         |--------------------------------------|
         |             Offset(2)                |
       5 |               K@STO2                 |
         └──────────────────────────────────────┘
```

Figure E-1. Symbol Table Entry

The end of a symbol table is indicated by a two-byte field that contains zeros.

An end-of-table entry in the symbol table is only one word long (halfword-aligned). It contains a pointer to the next segment of the symbol table, if any.

OBJECT CODE ADDRESS-LINE NUMBER TABLE

The object code address vs. line number table is at the end of the object code and is fullword-aligned. There is one entry in the table for each line in the source program on which a statement begins. Each entry is two words long. The first word contains an object code pointer and the second an integer line number in packed decimal. The terminating entry in the table contains an object code pointer consisting of the largest positive integer possible.

Thus, each statement of a CALL/360-OS PL/I source program and the object code generated for that statement are correlated. This serves as an important debugging aid. Figure E-2 shows the format of an entry in the table.

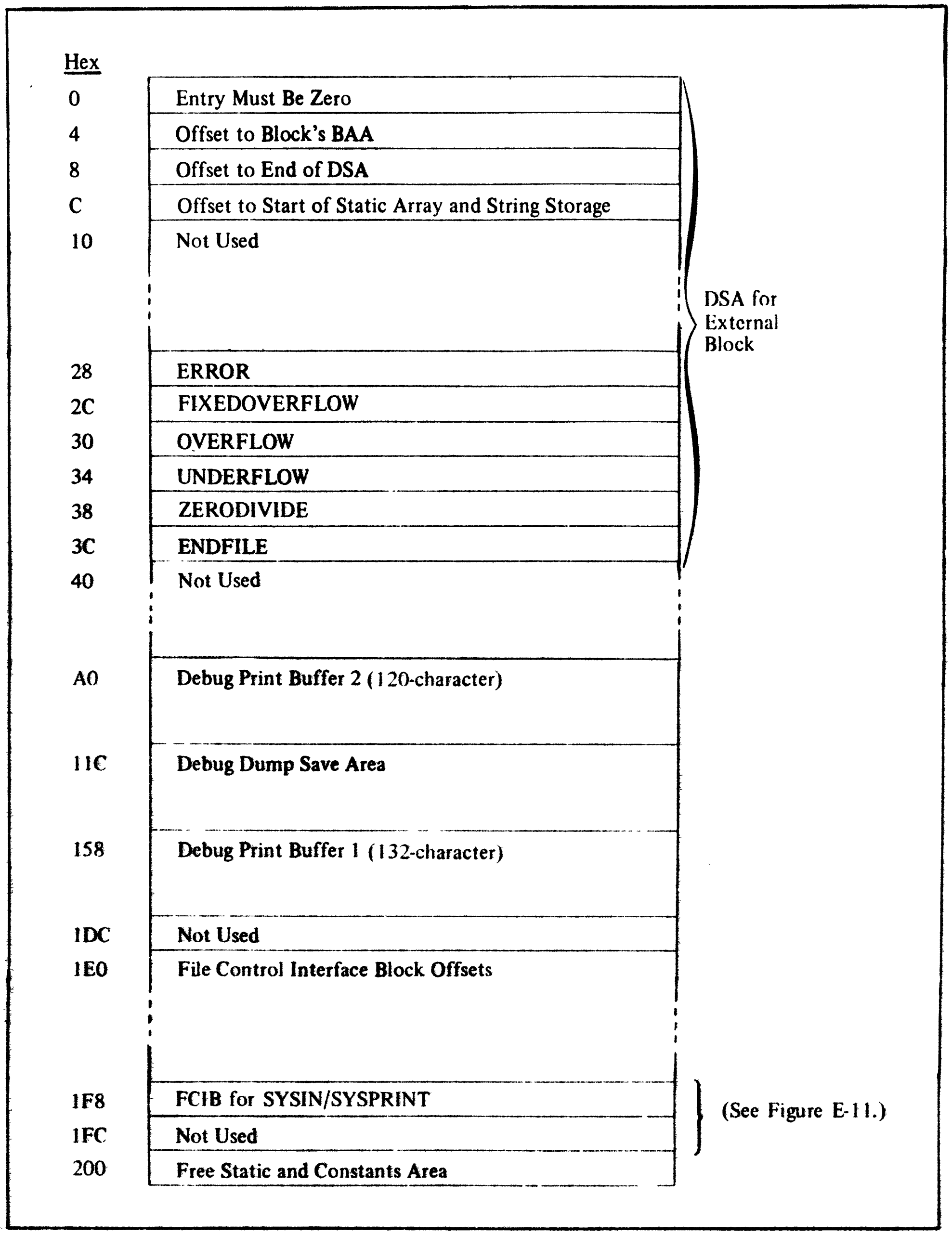
```
             1       2       3       4        Byte
         ┌──────────────────────────────┐
         |     Object Code Pointer      |
  Word 1 |           K@OLOP             |
         |------------------------------|
         | Line Number (Packed Decimal) |
       2 |           K@OLLN             |
         └──────────────────────────────┘
```

Figure E-2. Object Code Address-Line Number Entry

STATIC AND CONSTANTS STORAGE

This area contains all of the static storage, dope vectors, and constants used by the object program. Besides user-declared constants, data element descriptors (DED) and format element descriptors (FED) are included. This area actually consists of two subareas. The second of these areas contains the storage for all static arrays and strings. The first of these areas contains all other items. All automatic variables declared in the external procedure are treated as static variables.

The initial layout of the static and constants area is shown in Figure E-3.



Hex	
0	Entry Must Be Zero
4	Offset to Block's BAA
8	Offset to End of DSA
C	Offset to Start of Static Array and String Storage
10	Not Used
28	ERROR
2C	FIXEDOVERFLOW
30	OVERFLOW
34	UNDERFLOW
38	ZERODIVIDE
3C	ENDFILE
40	Not Used
A0	Debug Print Buffer 2 (120-character)
11C	Debug Dump Save Area
158	Debug Print Buffer 1 (132-character)
1DC	Not Used
1E0	File Control Interface Block Offsets
1F8	FCIB for SYSIN/SYSPRINT
1FC	Not Used
200	Free Static and Constants Area

Figure E-3. Static and Constants Area

Space for the six non-relocatable general purpose registers and the four floating-point registers is reserved in fixed locations at the beginning of this area. This space is used by the arithmetic interrupt instructions in the communications area and on-unit prologues and epilogues.

In order to make control of on-units in the external procedure the same as those in internal procedures, the first few words of this area are set up the same as the beginning of a dynamic storage area (DSA). Thus, in effect, static and constants storage is the DSA for the external procedure.

DATA ELEMENT DESCRIPTOR (DED)

This control block contains information derived from explicit and implicit declaration of variables of type arithmetic and string. DED formats are shown in Figure E-4, and the flag field of each DED is further described in Figure E-5.

Data Type	Representation	DED Formats (in bytes)		
		1	2	3
Arithmetic	Fixed-point Floating-point	Flags K@DDFF	P K@DDP	Q K@DDQ
String		Flags K@DDFF	Length	

Figure E-4. DED Formats

Bit	0	1	2	3	4	5	6	7	
0=	String	0	0	Fixed 0 Variable 1	1	1	0	0	
1=	Arith-metic	Inter-nal	0	Short	1	Decimal	Fixed	Real	=0
		Exter-nal	0	Long	1	Binary	Float	Complex	=1

• Figure E-5. Definition of DED Flag Field (K@DDFF)

The P Byte

P is the declared or default precision of the datum. The maximum values are 9 for fixed and 16 for float.

The Q Byte

Q is the declared or default scale factor of the datum, in excess-128 notation (that is, if the implied fractional point is between the last and next-to-last digit, Q will have the value 129).

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 four forms of the FED (all halfword-aligned):

1. Format Item E

```
    1        2        3        4        Byte
┌──────────────────────────────────┐
│      W        │   D    │   S      │
│    K@FEW      │ K@FED  │ K@FES    │
└──────────────────────────────────┘
```

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        Byte
┌───────────────────────────────────┐
│     W         │   D    │    P      │
│   K@FEW       │ K@FED  │  K@FEL    │
└───────────────────────────────────┘
```

W and D - (as for E-format)
P - scale factor in excess-128 notation

3. Format Items A and X

```
    1        2        Byte
┌─────────────────┐
│       W         │
│     K@FEW       │
└─────────────────┘
```

W - (as for E-format)

4. Printing Format Items SKIP and COLUMN

   The FED's for these format items are halfword binary integers.

DOPE VECTORS

String Dope Vector (SDV)

This control block specifies storage requirements for character-string data. An SDV consists of eight bytes (word-aligned), in the format shown in Figure E-6.

	1	2	3	4	Byte
Word 1	K@DVND=0	Byte Address of String (Offset) K@DVVO			
2 K@DVIN	ℓ		ℓ		

Figure E-6. SDV Format

where ℓ is the length-1 of the string in bytes; a negative value indicates the null string.

Array Dope Vector (ADV)

This control block contains information required in the derivation of elemental addresses within an array data aggregate. The ADV has three functions:

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.

In a CALL/360-OS PL/I implementation, arrays are stored in row-major order in storage. The elements of an array are normally in continuous storage; if the array is a cross-section, its elements may be discontiguous. Such discontiguity, however, is transparent to algorithms that employ an array dope vector.

The ADV contains (4+6n) bytes, where n is the number of dimensions of the array. The ADV is word-aligned. Its format is shown in Figure E-7.

```
              1         2         3         4         Byte
          ┌─────────────────────────────────────────┐
Word 1    | Number of  |     Virtual Origin          |
          | Dimensions |        K@DVVO               |
          |  K@DVND    |                             |
          |─────────────────────────────────────────|
     2    |                  |                      |
K@DVIN    | Multiplier(1)    | Multiplier(2)        |
          |                  |                      |
          |──────────────────+──────────────────────|
Word 3    |                  |                      |
     .    | . . .   . . .    | . . .   . . .        |
     .    |                  |                      |
     .    |──────────────────+──────────────────────|
          |                  |                      |
          | . . .   . . .    | Multiplier(n)        |
          |                  |                      |
          |──────────────────+──────────────────────|
          |                  |                      |
          | Upper Bound(1)   | Lower Bound(1)       |
          |                  |                      |
          |──────────────────+──────────────────────|
          |                  |                      |
          | . . .   . . .    | . . .   . . .        |
          |                  |                      |
          |──────────────────+──────────────────────|
          |                  |                      |
          | . . .   . . .    | . . .   . . .        |
          |                  |                      |
          |──────────────────+──────────────────────|
          |                  |                      |
          | Upper Bound(n)   | Lower Bound(n)       |
          |                  |                      |
          └─────────────────────────────────────────┘
```

Figure E-7. ADV Format

Definitions of ADV fields:

Virtual Origin - The byte address of the array element whose subscript values are all zero, that is, 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. (This address is an offset.)

Multiplier - Multipliers are halfword binary integers which in the standard ADV algorithm allow calculation of element addresses.

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. This value may be negative.

ADV Algorithm - Given subscript values for an n-dimensional array, the address of any element relative to the program origin is computed as:

$$\text{Address} = \text{virtual origin} + ((\ldots(S1*M1 + S2)*M2 + \ldots) + Sn)*Mn$$

where

Si = value of the ith subscript
Mi = number of different values the subscript in the (i+1)th dimension can assume, except for Mn
Mn = byte length of the element

String Array Dope Vector (SADV)

This control block contains information required to derive the address of elemental strings. The SADV is identical to the basic ADV, with the addition of a fullword to the end of the ADV which contains the length of the string in both halfwords (length-1 bytes and, if negative, the null string). (See Figure E-8.)

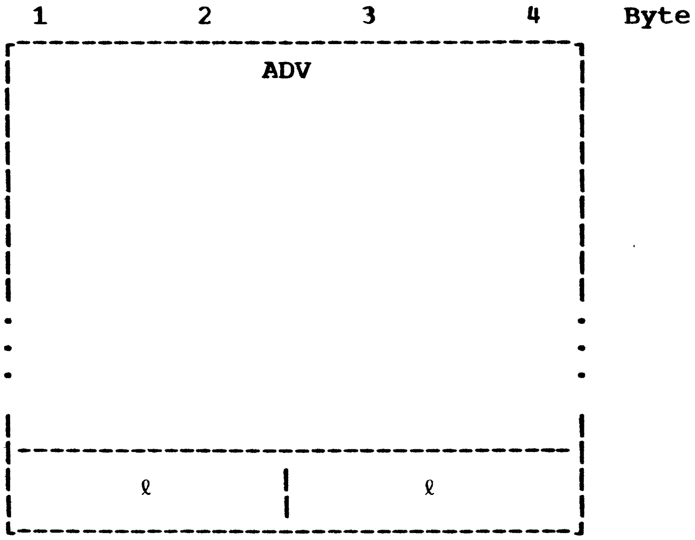
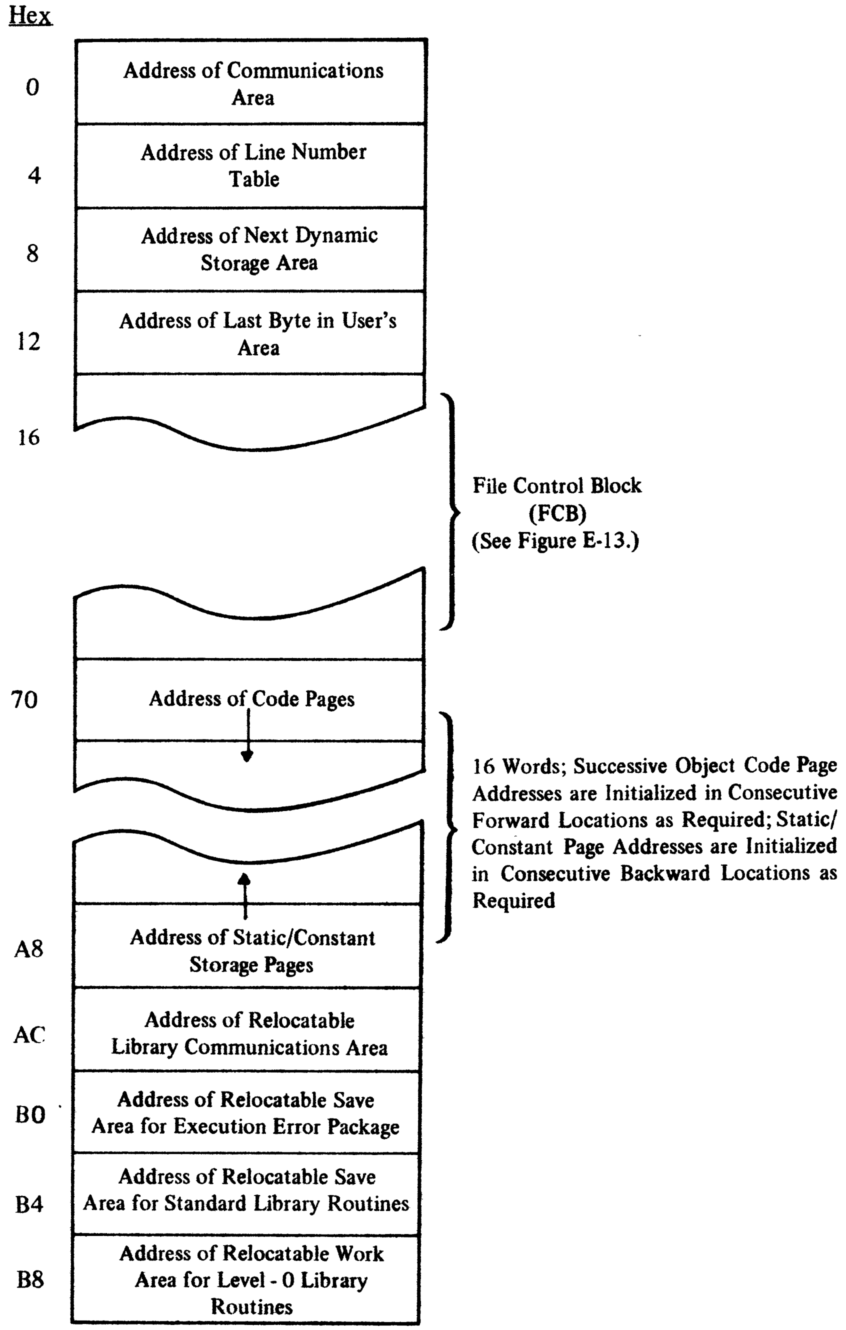


Figure E-8. SADV Format

ADDRESS CONSTANT AREA

The address constant (adcon) area is used during execution of the compiled object program to locate the data and library routines necessary to support execution. A symbolic description of this area is available to the compiler in the library load table (L table) so that symbolic references may be made to the area during code generation and compilation wrap-up.

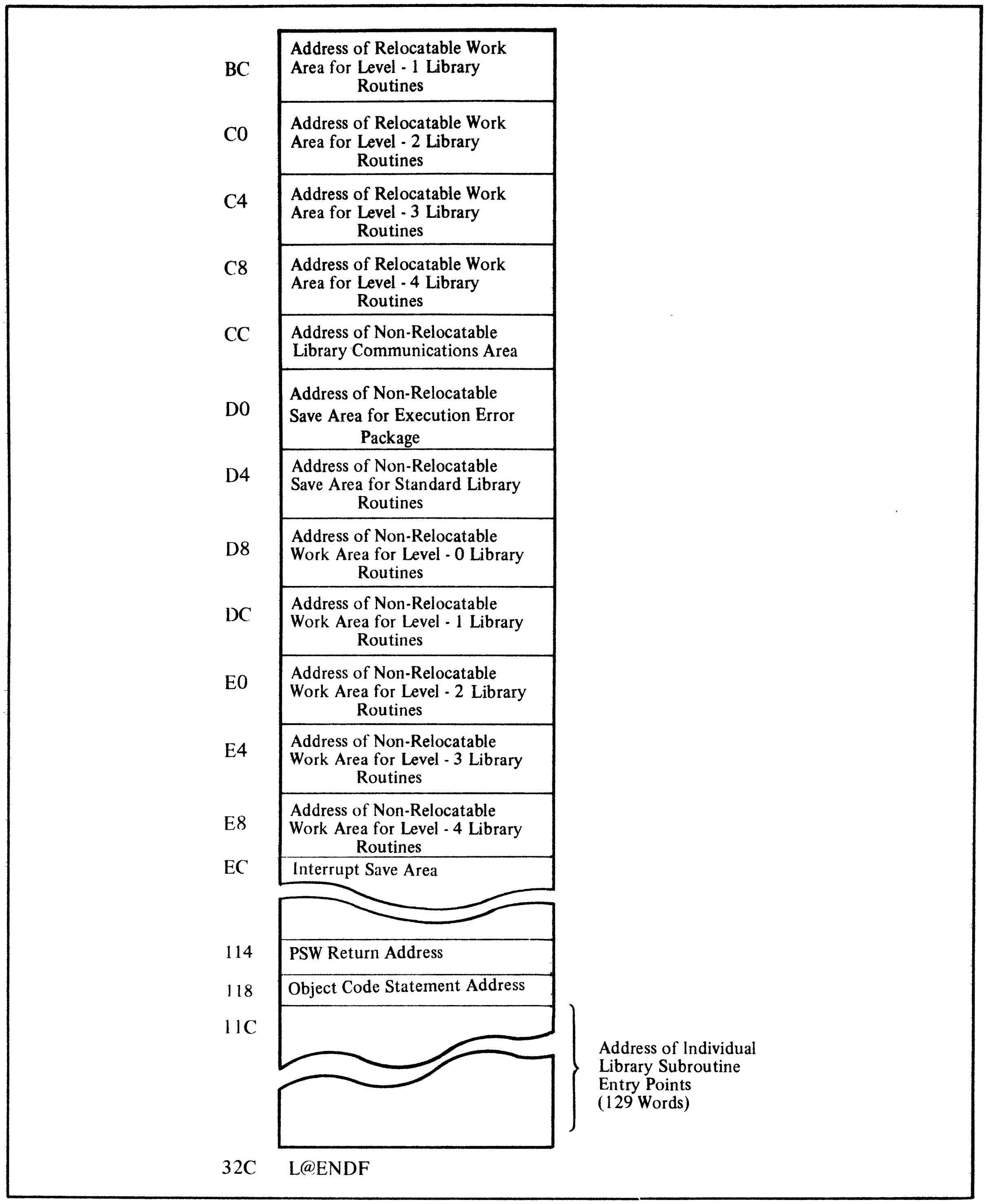
The adcon area consists of a fixed-length portion and a variable-length portion. The fixed-length portion of the area has the structure illustrated in Figure E-9.


Hex
0
Address of Communications Area
4
Address of Line Number Table
8
Address of Next Dynamic Storage Area
12
Address of Last Byte in User's Area
16
70
Address of Code Pages
A8
Address of Static/Constant Storage Pages
AC
Address of Relocatable Library Communications Area
B0
Address of Relocatable Save Area for Execution Error Package
B4
Address of Relocatable Save Area for Standard Library Routines
B8
Address of Relocatable Work Area for Level - 0 Library Routines

File Control Block (FCB) (See Figure E-13.)

16 Words; Successive Object Code Page Addresses are Initialized in Consecutive Forward Locations as Required; Static/ Constant Page Addresses are Initialized in Consecutive Backward Locations as Required

Figure E-9. Layout of Fixed-Length Portion of Adcon Area (Page 1 of 2)



BC	Address of Relocatable Work Area for Level - 1 Library Routines
C0	Address of Relocatable Work Area for Level - 2 Library Routines
C4	Address of Relocatable Work Area for Level - 3 Library Routines
C8	Address of Relocatable Work Area for Level - 4 Library Routines
CC	Address of Non-Relocatable Library Communications Area
D0	Address of Non-Relocatable Save Area for Execution Error Package
D4	Address of Non-Relocatable Save Area for Standard Library Routines
D8	Address of Non-Relocatable Work Area for Level - 0 Library Routines
DC	Address of Non-Relocatable Work Area for Level - 1 Library Routines
E0	Address of Non-Relocatable Work Area for Level - 2 Library Routines
E4	Address of Non-Relocatable Work Area for Level - 3 Library Routines
E8	Address of Non-Relocatable Work Area for Level - 4 Library Routines
EC	Interrupt Save Area
114	PSW Return Address
118	Object Code Statement Address
11C	Address of Individual Library Subroutine Entry Points (129 Words)
32C	L@ENDF

• Figure E-9. Layout of Fixed-Length Portion of Adcon Area (Page 2 of 2)

The variable-length portion contains adcons for a class of library subroutines which require individual block adcon areas. The variable-length portion also contains library routine parameter lists and the adcon portion of the library work space (LWS). Only adcons for those routines actually required by the compilation are included in the variable-length portion.

MULTI-FILE INTERFACE

The interface for terminal and disk files involves pointers in the communications area, tables in the static and constants area, file control interface blocks (one per filename) located in the static and constants area, and a file control block in the fixed-length portion of the adcon area.

COMMUNICATIONS AREA

The communications area is used for communication with the Executive during I/O operations. Its contents are shown below.

```
                  Hex
                             ┌──────────────────────────────────────────┐
    FILEPTR       11C        │  (File Table Offset From Commun) (1E0)   │
                             └──────────────────────────────────────────┘
                             ┌────────────────────┐
    FILENBR       122        │  File Index        │  (1,2,3, or 4)
                             └────────────────────┘
```

Figure E-10. Communications Area

STATIC AND CONSTANTS AREA

FCIB Offsets and FCIB's for SYSIN and SYSPRINT

Up to four disk files can be open at one time. The FCIB's for the open disk files are pointed to by the first four words of the area shown in Figure E-11. There are no pointers to the FCIB's for SYSIN and SYSPRINT. Their origins are fixed as the first and second FCIB pointer words. This overlay is feasible because only the locations of the buffer pointer words of the FCIB's for SYSIN and SYSPRINT are valid. These pointers are the last two words of the area and point to displacements 2C and 34 of the adcon area. (See "Adcon Area (Fixed-Length Portion).")

	Hex		
FILETABL	1E0	A (FCIB Offset (From Commun))	*
	1E4	A (FCIB Offset (From Commun))	**
	1E8	A (FCIB Offset (From Commun))	
	1EC	A (FCIB Offset (From Commun))	
	1F0	End of Table (FFFF)	
		Unused	
	1F8	00 A(SYSIN Buffer Pointer Pair) 2C	
	1FC	81 A(SYSPRINT Buffer Pointer Pair) 34	

* - Start of Dummy SYSIN FCIB
** - Start of Dummy SYSPRINT FCIB

Figure E-11. FCIB Offsets and FCIB's for SYSIN and SYSPRINT

FCIB's for Disk Files

An FCIB is built in the static and constants area for each filename used. This area is defined by FCIBDEF DSECT in the Zap macro (IHEZAP). The format of each FCIB is shown in Figure E-12.

```
           Hex

                  r-----------------------------------------------------------1
FCIBTITL    0     |                                                           |
                  |                          TITLE                            |
                  |                                                           |
            4     |                                             r-------------|
                  |                                             |             |
                  |                                             |OPEN/Mode    |
            8     |                                             |             |
FCIBMODE          |---------------------------------------------|-------------|
                  | I/O Code  |Record| Half-Tracks              |Max. Half-   |
FCIBEXEC    C     |           |Number| Allocated                |Tracks       |
                  |           |      |                          |Allowed      |
                  |-----------------------------------------------------------|
FCIBNAME    10    |                                                           |
                  |                        FILENAME                           |
                  |                                                           |
            14    |-----------------------------------------------------------|
                  | File       | Offset from Adcon Area of                    |
                  | Codes      | Buffer Pointer Pair                          |
FCIBFC      18    |------------+----------------------------------------------|
FCIBFCB           |            |                                              |
                  | FILENBR    |                                              |
FCIBNBR     1C    L-----------------------------------------------------------J
```

Figure E-12. FCIB Format for Disk Files

File Codes:

Bits 0 and 1 =	00	SYSIN (terminal)	Declared file types
	01	Disk input	
	10	SYSPRINT (terminal)	
	11	Disk output	
2 =	1	Disk input	Set by OPEN statement
3 =	1	Disk output	
4 =	1	List or data I/O flag (disk environment in compiler)	
5 =	0	Not busy	
	1	Busy	
6 =	0	External format	
	1	Internal format	
7 =	0	Non-print file	
	1	Print file	

ADCON AREA (FIXED-LENGTH PORTION)

The FCB in this area contains a common data specification set up for the currently active file and six pairs of buffer pointer words. It is defined by the FCIBDEC DSECT in the Zap macro (IHEZAP) as shown in Figure E-13.

Hex			
10	Codes	Start of Buffer	Common Data Specification
14		Current Buffer Position	
18			
1C			
20			
24			
28	FILENBR	Address of FCIB	
2C	Codes	Start of Buffer	SYSIN
30		Current Buffer Position	
34	Codes	Start of Buffer	SYSPRINT
38		Current Buffer Position	
3C	Codes	Start of Buffer	Disk Input or Output
40		Current Buffer Position	
44	Codes	Start of Buffer	
48		Current Buffer Position	
4C	Codes	Start of Buffer	
50		Current Buffer Position	
54	Codes	Start of Buffer	
58		Current Buffer Position	

Figure E-13. FCB Format in Fixed Adcon Area

File Codes:

Bits 0 and 1 =	00	SYSIN (terminal)
	01	Input (disk)
	10	SYSPRINT (terminal)
	11	Output (disk)
2 =	0	Not used
3 =	0	Not used
4 =	1	List or data I/O flag
5 =	0	Not busy
	1	Busy

```
6 =   0  External format
      1  Internal format
7 =   0  Non-print file
      1  Print file
```

Common Data Specification Portion of FCB

Issuance of a GET or PUT statement causes the common data specification portion of the FCB (words 1 through 7) to be set. The buffer pointer words (words 1 and 2 of the area) are obtained from either the pair for SYSIN or SYSPRINT or a pair set up for an open disk file. The location of the proper buffer pointer pair in the FCB is indicated by a pointer in the FCIB of the file referenced in the GET or PUT statement.

Byte 1 of word 7 is set with the file index also set in FILENBR of the communications area. The remainder of word 7 contains the address of the FCIB. The contents of words 3 through 6 depend on the type of I/O being performed. Various possibilities are shown in Figures E-14 through E-19.

```
             1       2       3       4      Byte
          ┌─────────────────────────────────┐
  Word 3  |                                 |
          |                                 |
          |----           Not            ---|
       4  |                                 |
          |                                 |
          |----           Used           ---|
       5  |                                 |
          |                                 |
          |---------------------------------|
       6  |    Address of Symbol Table      |
          |            K@FBST               |
          └─────────────────────────────────┘
```

Figure E-14. Common Data Specification Portion of FCB for Data Input and Non-Array Element Data Output

```
             1       2       3       4      Byte
          ┌─────────────────────────────────┐
  Word 3  |        Address of Element       |
          |             K@FBEL              |
          |---------------------------------|
       4  |                                 |
          |                                 |
          |----           Not            ---|
       5  |                                 |
          |               Used              |
          |---------------------------------|
          |    Address of Symbol Table      |
       6  |            K@FBST               |
          └─────────────────────────────────┘
```

Figure E-15. Common Data Specification Portion of FCB for Array Element Data Output

```
            1       2       3       4       Byte
         ┌───────────────────────────────────┐
  Word 3 │      Address of Skip Value        │
         │             K@FBSK                │
         │───────────────────────────────────│
       4 │                                   │
         │                                   │
         │----           Not              ---│
       5 │                                   │
         │                                   │
         │----           Used             ---│
       6 │                                   │
         │                                   │
         └───────────────────────────────────┘
```

Figure E-16. Common Data Specification Portion of FCB for Initialize Output with SKIP Option

```
            1       2       3       4       Byte
         ┌───────────────────────────────────┐
  Word 3 │      Address of Element           │
         │           K@FBEL                  │
         │───────────────────────────────────│
       4 │      Address of DED               │
         │           K@FBDD                  │
         │───────────────────────────────────│
       5 │                                   │
         │                                   │
         │----           Not              ---│
       6 │               Used                │
         │                                   │
         └───────────────────────────────────┘
```

Figure E-17. Common Data Specification Portion of FCB for List I/O

```
            1       2       3       4       Byte
         ┌───────────────────────────────────┐
  Word 3 │        Address of Element         │
         │             K@FBEL                │
         │───────────────────────────────────│
       4 │        Address of DED             │
         │             K@FBDD                │
         │───────────────────────────────────│
       5 │        Address of FED             │
         │             K@FBFE                │
         │───────────────────────────────────│
       6 │           Not Used                │
         │                                   │
         └───────────────────────────────────┘
```

Figure E-18. Common Data Specification Portion of FCB for Non-Complex Edit I/O

	1	2	3	4	Byte
Word 3	Address of Element K@FBEL				
4	Address of DED K@FBDD				
5	FED Type K@FBFT	Address of Real FED K@FBFE			
6	FED Type K@FBIT	Address of Complex FED K@FBIF			

Figure E-19. Common Data Specification Portion of FCB for Complex Edit I/O

BLOCK ADCON AREA

Each procedure and begin block in the program has a block adcon area (BAA) in adcon storage. The BAA contains all the information needed by the block. For a begin block, the area is four words long. For a procedure block, it is six words plus one word for each parameter. The format of the BAA for every block other than the external block is given in Figure E-20.

The BAA for the external block is pointed to by the second word of the static and constants area. The location pointed to is at displacement A4 from the start of the fixed adcon area. (Refer to Figure E-9.)

```
              r-----------------------------------------------------------\
      /Word 1 | Number of |     Address of Block Entry Point              |\
     |        | Parameters|              K@BAEP                           | |
     |        |  K@BANP   |       (Not Used for Begin Blocks)             | |
      \       |-------------------------------------------------------------| |
       |    2 |               Address of Block's DSA                        | |
Begin  |      |                       K@BADS                                | |
Block <       |-------------------------------------------------------------| |
       |    3 |         Address of Location Following DSA                   | |
       |      |                       K@BAED                                | |
      /       |-------------------------------------------------------------| |
     |      4 |               Block's Epilogue Address                      |  \
     |        |                       K@BAEL                                |   >Procedure
     |        |-------------------------------------------------------------|  / Block
      \     5 |                   Return Address                            | |
              |                       K@BARA                                | |
              |-------------------------------------------------------------| |
              |                                                             | |
   K@BAPM---> .         Address of Arguments as Stored by Code               . |
              .         Generated for CALL or Function Reference             . |
              .                                                              . |
              |                                                             | |
              |-------------------------------------------------------------| |
              |                                                             | |
              |          Address of Return Variable (see note)              |/
              L-------------------------------------------------------------J
```

Note: This field is significant only if the routine is referenced as a function (rather than called). It is set up by code generated for the function reference.

Figure E-20. Format of Block Adcon Area (BAA)

ON-UNIT ADCON AREA

The general format of an ON statement is:

option 1. ON-condition on-unit
option 2. ON-condition SYSTEM;

During compilation, space for an on-unit adcon area is allocated for each ON statement of the form shown in option 1. The format of the on-unit adcon area for all on-units except ON ENDFILE is illustrated in Figure E-21.

Word 1	Address of On-Unit Entry Point K@BAEP
2	Address of DSA K@BADS
3	Address of Following DSA K@BAED
4	Not Used
5	PSW Save Word (Second Word of PSW) K@ONPS
K@ONSA---> 6-15	Relocatable Register Save Area (Registers 6-15)
16-75	Save Area for Relocatable Registers from Levels 0, 1, and 2 Work Space (240 bytes)

Figure E-21. Format of On-Unit Adcon Area (Except for ON ENDFILE)

Word 1 - the location of the code that will be performed if the ON-condition is raised.

Word 2 - the address of the DSA (on-unit format) obtained when the ON-condition is raised.

Word 3 - pointer to the word following the last word in the DSA pointed to by word 2.

Word 4 - Not used.

Word 5 - the second word of the PSW, which is saved when the ON-condition is raised.

Words 6 through 15 - If the ON-condition is raised, the relocatable registers are saved in these locations. The non-relocatable registers are saved in the DSA pointed to by word 2.

Words 16 through 75 - If the ON-condition is raised, the relocatable sections of the level 0, level 1, and level 2 save areas are moved to these locations. The non-relocatable save areas of these levels are moved to the DSA pointed to by word 2.

The format of the on-unit adcon area for ON ENDFILE is illustrated in Figure E-22.

	Dec	Hex	
ENDCON1	0	0	Address of On-Unit Entry Point
ENDCON2	4	4	Address of DSA
ENDCON3	8	8	Address of Word Following DSA
ENDCON4	12	C	Not Used
ENDCON5	16	10	PSW Save Word (Second Word of PSW)

Figure E-22. Format of ON ENDFILE Adcon Area

The ON ENDFILE adcon area has the same format as words 1 through 5 of the on-unit adcon area for other types of on-units. The contents of these words have similar meanings.

LIBRARY

At the beginning of the library area is the non-adcon portion of the library work space. Immediately following are all of the library routines needed for the object program.

STATIC ARRAY AND STRING STORAGE

Space for arrays and strings declared in the external block is allocated in this region. All items in this area are referenced by a dope vector in the static and constants area.

DYNAMIC STORAGE AREAS AND ON-CONDITIONS

A DSA (block type) is obtained during the initialization process for internal procedures and begin blocks. (The external procedure block is assigned an area within the static and constants area which serves the function of a DSA and is thus called the external block's DSA.) In addition, a DSA (on-unit type) is obtained if an ON-condition covered by an option-1 ON statement occurs (see "On-Unit Adcon Area", above). The space obtained for these DSA's is released when the block is exited or when the code specified by the option-1 ON statement has been executed.

CALL/360-OS PL/I ON-conditions are error, fixed-point overflow, other overflow conditions, underflow, zerodivide, and end of file. Each DSA for a procedure or begin block contains ERROR, FIXEDOVERFLOW, OVERFLOW, UNDERFLOW, ZERODIVIDE, and ENDFILE words corresponding to these conditions.

The format of each of the first five interrupt condition words is shown below.

Action Code	Pointer to On-Unit Adcon Area

The format of the ENDFILE word follows.

Number of Entries	Pointer to ENDFILE Table

The first byte of the ENDFILE word indicates the number of entries in the ENDFILE table. (See Appendix B.)

There is an entry in the ENDFILE table for each unique file referenced within a block containing an ON ENDFILE statement. The first word of that entry and the DSA words for the interrupt conditions have the same format. They are often called action words.

Part of the initialization for a block is to set up the ENDFILE table and to set all action words to zero. Execution of any ON statement causes the setting up of an action word. The meaning of the action word is determined by the action code byte. Code values are explained below.

<u>Code</u> <u>Meaning</u>

0 Either an ON statement for this condition or file has not been executed in this block or a REVERT statement was the last statement executed for this condition or file. If the corresponding ON-condition is raised, the code byte of the corresponding word in the immediately preceding DSA will be checked. If 1 or 3, the action indicated by this code will be performed. Otherwise, the next preceding DSA will be checked. This process will continue until either a code byte equal to 1 or 3 is found or all preceding DSA's have been searched. In the latter case, the standard system action will then be performed.

1 This code value is set by the execution of an option-2 ON statement or by execution of a REVERT statement which resets conditions to those specified by a previous ON SYSTEM statement. If the corresponding ON-condition is raised, the standard system action will be performed.

3 When an option-1 ON statement or a REVERT statement which resets conditions to those specified by a previous option-1 ON statement is executed, the action-code byte is set to 3. The last three bytes are set to point to the on-unit adcon area for that statement. If the corresponding ON-condition is raised, the action specified by the ON statement will be performed after an on-unit DSA is obtained. If the ON-condition was not ON ENDFILE, registers will be saved in the on-unit adcon area and DSA area. Levels 0, 1, and 2 work areas will be moved to the adcon area and DSA area.

Figure E-23 illustrates the DSA for internal procedure and begin blocks. The ENDFILE table is pointed to by the immediately preceding ENDFILE word.

```
                Dec     Hex

                                 ┌──────────────────────────┐
ENDDSA1          0       0       | Pointer to Previous DSA  |
                                 |--------------------------|
ENDDSA2          4       4       | Pointer to BAA           |
                                 |--------------------------|
ENDDSA3          8       8       | Unused                   |
                                 |--------------------------|
ENDDSA4         12      20       | Unused                   |
                                 .                          .
                                 .                          .
                                 |--------------------------|
ENDDSA5         40      28       | ERROR                    |
                                 |--------------------------|
                                 | FIXEDOVERFLOW            |
                                 |--------------------------|
                                 | OVERFLOW                 |
                                 |--------------------------|
                                 | UNDERFLOW                |
                                 |--------------------------|
                                 | ZERODIVIDE               |
                                 |--------------------------|
ENDDSA6         60      3C       | ENDFILE                  |
                                 |--------------------------|
ENDDSA7         64      40       | ENDFILE TABLE            |
                                 .                          .
                                 .                          .
                                 .                          .
                                 |--------------------------|
                                 . Automatic Arithmetic     .
                                 . Scalars, Strings, and    .
                                 . Arrays                   .
                                 └──────────────────────────┘
```

Figure E-23. Layout of DSA for Internal Procedure and Begin Blocks

ENDDSA1 is a pointer to the previous DSA (which is either that of the next outer block or that of an on-unit). ENDDSA2 is a pointer to the block adcon area (BAA). Automatic arithmetic scalars, strings, and arrays are set up at initialization time. (For a discussion of the contents of ENDDSA5, ENDDSA6, and ENDDSA7, see preceding paragraphs.)

A portion of the static and constants area is called the DSA for the external block. However, this storage area is not dynamic. It also differs from other DSA's in that the ENDFILE table is not adjacent to the ENDFILE word. The ENDFILE table is in the free static and constants area of the static and constants area.

Figures E-24 and E-25 illustrate the DSA areas for on-units.

Dec	Hex	
0	0	Pointer to Previous DSA
4	4	Pointer to On-Unit Adcon Area
8	8	General Purpose Registers G0-G5
32	20	Floating-Point Register F1
40	28	Set to All Zeros
60	3C	
64	40	Floating-Point Registers F2-F4
88	58	Level 0 Non-Relocatable Area (first 120 bytes)
208	D0	Level 1 Non-Relocatable Area (first 120 bytes)
328	148	Level 2 Non-Relocatable Area (first 120 bytes)

Figure E-24. Layout of DSA for On-Units (Except ON ENDFILE)

	Dec	Hex	
ENDDSA1	0	0	Pointer to Previous DSA
ENDDSA2	4	4	Pointer to On-Unit Adcon Area
ENDDSA3	8	8	Unused
ENDDSA4	12	20	Unused
ENDDSA5	40	28	Set to All Zeros
ENDDSA6	60	3C	

Figure E-25. Layout of DSA for ON ENDFILE On-Units

EXAMPLES

In summary, a few examples are given below.

Example 1: Assume that a call is made to internal procedure INT1 (a portion of which is shown below).

```
100  INT1:  PROCEDURE;
         .
         .
         .
200  ON UNDERFLOW SYSTEM;
         .
         .
         .
```

```
300  Z = 5+10**-40/10**-42;
          .
          .
          .
900  END;
```

A DSA (block type) is initialized. Then, the following actions occur.

1. As part of INT1, the ON UNDERFLOW SYSTEM; statement is executed. This causes the action-code byte of the UNDERFLOW word in the DSA for the internal block (that is, for INT1) to be set to one.

2. The code generated for the assignment statement causes an attempt to divide 10**-40 by 10**-42. A machine interrupt occurs.

3. The Executive transfers control to the Error Routine (IHEERR) via the code at ARINTRP of the communications area. IHEERR determines that the interrupt is due to an underflow condition.

4. The action-code byte of the UNDERFLOW word in the DSA is examined and found to be one. Therefore, standard system action (printing of the UNDERFLOW message) is performed.

5. Return is made to the point of interrupt.

<u>Example 2</u>: Assume that a call is made to internal procedure INT2 (a portion of which is shown below).

```
100  INT2:  PROCEDURE;
          .
          .
          .
200  OPEN FILE(FIHL2) INPUT;
          .
          .
          .
300  ON ENDFILE(FIHL2) X=2;
          .
          .
          .
390  DO I = 1 to 10;
400  GET FILE(FIHL2) A,B,C;
          .
          .
          .
800  END;
900  END;
```

A DSA (block type) is set up. Then, the following actions occur.

1. As part of INT2, the ON ENDFILE(FIHL2) X=2; statement is executed. This causes the action-code byte of the action word of the ENDFILE table entry in the DSA that corresponds to FIHL2 to be set to 3. The remaining three bytes of the action word are set to point to the on-unit adcon area for this statement (which was allocated when the statement was compiled). Note that X=2 is not executed.

2. Assume that FIHL2 contains only nine sets of items to be read into A, B, and C. Then, the tenth execution of the GET statement causes an attempt to read past the last data item on FIHL2.

3. The ENDFILE condition is recognized by the List- and Data-Directed Input routine (IHELDIB). It calls IHEERR.

4. IHEERR examines the action-code byte of the action word in the ENDFILE table entry for FIHL2. Since the byte contains a code of 3, IHEERR performs the following actions.

   a. Initializes the on-unit adcon area pointed to by the action word.

   b. Obtains main storage locations for an on-unit DSA.

   c. Initializes the on-unit DSA.

   d. Transfers control to the on-unit code pointed to by the first word of the adcon area. This code corresponds to X=2 and causes X to be set to 2.

5. The on-unit code transfers control to entry-point IHEERRN of Error Routine (IHEERR).

6. IHEERR releases the on-unit DSA area and transfers control to the next statement in the internal block (that is, in INT2).

<u>Example 3</u>: Assume that a call is made to internal procedure INT3 (a portion of which is shown below).

```
100  INT3:  PROCEDURE;
             .
             .
             .
200  ON UNDERFLOW BEGIN;
210  Z=0;
220  SWT2=5;
230  END;
300  X=5;
             .
             .
             .
400  R=F/Y;
             .
             .
             .
900  END;
```

A DSA (block type) is set up. Then, the following actions occur.

1. As part of INT3, the ON UNDERFLOW BEGIN; statement is executed. This causes the action-code byte of the UNDERFLOW word in the DSA for internal procedure INT3 to be set to 3. The remaining bytes of the UNDERFLOW word are set to point to the on-unit adcon area (established for this statement at compile-time).

2. Control is transferred past the code generated for the begin block (to statement 300).

3. A machine interrupt occurs while the expression R=F/Y is being computed. Control is passed to IHEERR.

4. IHEERR determines that the interrupt is due to an underflow condition. Then it determines that the action-code byte of the UNDERFLOW word is set to 3. As a result, IHEERR performs the following actions.

   a. Saves relocatable registers and second word of PSW in the on-unit adcon area pointed to by the rightmost three bytes of the UNDERFLOW word.

b. Moves levels 0 through 2 relocatable library work space to on-unit adcon area.

c. Gets a DSA for the on-unit and saves non-relocatable registers in this area.

d. Moves levels 0 through 2 non-relocatable library work space to the DSA.

e. Transfers control to the code generated for the begin block via the first word of the on-unit adcon area.

5. The initialization code for the begin block causes another DSA (block type) to be generated. In addition, the block may contain option-1 ON statements (that is, having specified on-units) that cause action words to be set up for this DSA when the ON statements are executed.

6. After the main code of the begin block has been executed, its epilogue code is performed. The DSA for the begin block is released.

7. Control is transferred to entry-point IHEERRR of Error Routine (IHEERR).

8. IHEERR performs the following actions.

    a. Restores fixed and floating-point registers. Restores PSW.

    b. Restores levels 0, 1, and 2 library work space.

    c. Releases the DSA for the on-unit.

    d. Causes control to be returned to the point of interrupt.

DATA ADDRESSING

All items in the object program can be addressed by a combination of a base address and a displacement from that address. All necessary base addresses are either in the adcon area or permanently assigned to relocatable registers. Thus, the base address can be easily obtained and the displacement added to give the true address of the item.

The object program uses ten relocatable registers, six of which have permanently assigned values. The other four (12 to 15) are used for obtaining necessary base addresses and for linkage. Figure E-26 shows the contents of the general purpose registers.

			Library Designation
GPR	0-5	Fixed-point arithmetic	G0 to G5
	6-8	Code cover (first 12,288 bytes)	P0 to P2
	9	Adcon area cover	P3
	10	Static and constants cover	P4
	11	Current DSA cover	P5
	12	Volatile	P6
	13	Volatile and parameter list cover	P7
	14	Volatile and return address	P8
	15	Volatile and entry point address	P9

Figure E-26. General Purpose Register Assignment

During subroutine linkage, the parameter list register and the entry point register are used only if needed. Subroutine linkage assumes that general purpose registers 2 through 11 and floating-point registers 4 and 6 are the same upon return.

APPENDIX F - SUPPORT SERVICES FOR LANGUAGE PROCESSORS

The CALL/360-OS PL/I compiler (language processor) runs in a simplified time-sharing environment under the control of the CALL/360-OS Executive. Facilities are simplified in keeping with the design objective of a high-performance system. The CALL/360-OS Executive analyzes and responds to all terminal commands. It provides a line editor that accepts source programs from a terminal and arranges this input for compilation.

The interface between the CALL/360-OS Executive and its associated processors and user programs is based on the following requirements:

- All processors and user programs are relocatable.
- All processors are reentrant.
- All jobs can be described for the present as the sequence: compilation plus execution.
- At compilation time, two modules are in use by the terminal: the processor and the user program area. The processor is considered to have control.
- At execution time, only one module is in use by the terminal: the user program area. All runtime I/O routines, arithmetic functions, etc., are attached to the program area as a runtime package. The user program is considered to have control.
- Control may be taken away from the language processor or user program at any time (with two exceptions--see below) and the user program area written onto the disk.
- This generally happens when a program uses its initial time quantum (presently about three seconds). When it is time for the user to "get another time slice," the user area is read from the disk into (probably) a different area of core. This process is generally referred to as a "time-quantum swap."

The interface is designed so that a minimum amount of interaction is needed. This is a necessary feature in a time-sharing system where 80-90 percent of all jobs are executed in less than 900 milliseconds. The interface consists primarily of two core communications regions. The regions are the communications area of the user work area and the user terminal table.

The CALL/360-OS Executive can be called only via the SVC instruction in the communications area. To issue a request to the Executive, a language processor or runtime program must load register 0 with a request code and then execute the SVC instruction to transfer control to the Executive. In CALL/360-OS PL/I, this is accomplished at compile time by calling the SVC Director ($SVC); it is accomplished at runtime by calling the Library SVC Director (IHESVC). Either routine loads register 0 with the parameter passed by the calling instructions and executes the SVC. The code in register 0 tells the CALL/360-OS Executive what action to take. A language processor on this system should not exceed 81,920 bytes (forty 2048-byte blocks) in order to achieve effective utilization of memory.

Note: In CALL/360-OS documentation, the request code loaded in register 0 is usually referred to as an SVC code.

COMPILER/EXECUTIVE INTERACTIONS

STORAGE ALLOCATION

To eliminate the necessity for elaborate and time-consuming core management routines, when a user specifies that his program is to be compiled and executed, the space necessary to accomplish this is allocated in one contiguous block. This block contains, at various stages in compilation and execution, the communications area, source program, object program, compiler work space, disk and terminal I/O buffers, etc. The Executive uses a unique core allocation algorithm for each processor in the system. This algorithm must be expressed in terms of constants and the following variables:

1. Number of bytes in the source program

2. Number of lines in the source program

The algorithm should be such that compilation and execution of at least 90 percent of all programs using that processor can be accomplished within the allocated space. The actual core area allocation is the smallest number of 2048-byte blocks which completely contain the computed number of bytes.

There is a 2048-byte area at the very bottom of the user area that the Executive uses for holding various pointers. This area is swapped with the user program. Language processors, however, are generally unaware of its existence.

The allocated area may be arranged by the processor in any way with the following restrictions:

1. A communications area must exist at the bottom (that is, in the lower-numbered locations) of every user area.

2. Before compilation, the source program will be placed by the Executive at the top of the user area, at a location indicated in the user terminal table.

If, during the course of compilation, it is determined that the amount of core initially allocated is insufficient, the additional core required is requested by the SVC in one of two modes. The first mode is used when compilation has been completed, and the Executive need only attach the extra core to the existing area and return control to the requester. The second mode of the SVC is used when compilation is incomplete, and the amount of extra core required is indeterminate. In this case, the Executive will add a percentage of the original allocation to the area, set a bit in the UTT indicating that reallocation has taken place, and restart the compilation from the beginning.

INITIAL REGISTER SETTINGS

Before passing control to a processor, the Executive sets register 7 with the base of the processor, and register 12 with the base of the user (program) area. Control is then passed to the first byte of the processor.

USER WORK AREA

The Executive places the source program entered by the user at the end (higher-numbered locations) of the user work area. This text contains line numbers and end-of-line indicators. The format of the source lines is depicted in Figure F-1.

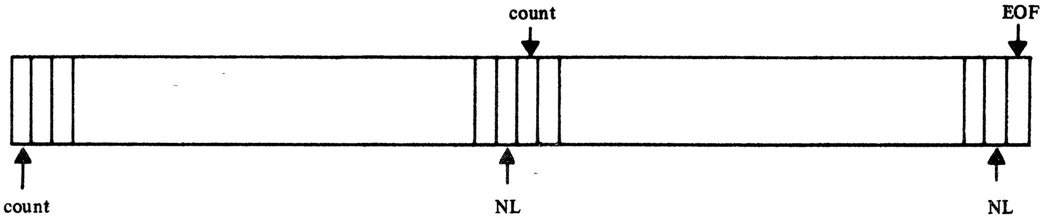


Figure F-1. Format of CALL/360-OS Source Lines

Each line is started by a count byte. The count is in binary. This byte contains the number of bytes in the line including the count byte itself. The next character after the count byte is a numeric character which is the first character of the line number. The line number is one to five numeric characters in length and is terminated by the first nonnumeric character if the number is less than five digits. The last character in the line is the new line (NL) character.

Source lines begin and end on byte boundaries. There are no spaces or fills between lines. The last character of the source program is an EOF character which is hex 01. The EOF character is in the position occupied by what would be the count byte if there were another line.

The beginning (lower-numbered locations) of the user work area has the user communications area. The communications area is used to pass parameters between the compiler and the object program. Some of the items contained in this communications area are:

1. Pointer to and length of address constant area:

   a. For compiler's address constants

   b. For user's address constants

2. Register relocation information:

   a. For registers containing compiler address constants

   b. For registers containing user address constants

3. Interrupt control information

4. Swap flag

5. Terminal I/O buffer

6. Pointer to next available byte in terminal I/O buffer

7. Terminal output inhibit flag

8. Pointer to user's UTT entry (set by the CALL/360-OS Executive)

A complete list of the contents of the communications area is given under "Communications Area" in this appendix.

USER TERMINAL TABLE

The user terminal table (UTT) is a table that is maintained by the CALL/360-OS Executive in its own area. It is primarily for the use of the Executive. However, it also contains information that is needed by the compiler. It may be read, but not written, by the compiler. Some of the items contained in the UTT are:

1. Length of user's work area

2. Location and length of source program

3. Information as to whether disk files have been attached

The contents of the UTT required by the compiler are listed under "UTT Data Available to Language Processor" in this appendix.

The UTT is assembled as a DSECT macro and is available to all processor writers. It will be supplied either on cards or through the macro library.

ADDRESSING

All processors and user programs in the system must be interruptable and relocatable at any time (with two exceptions which are described below). When a processor or user program is relocated, the Executive will update all 24-bit addresses referencing the relocated area before returning control to the point of the interrupt. All updating will be made by performing fullword adds. To make this possible, the location of these values must be specified in the communications area of the user program. This is done by means of six words (CSPTR through PSREG): three describing those values to be relocated with the processor and three describing those to be relocated with the user program. The first word in each set points to the beginning of the contiguous block containing that type of saved value. The second word indicates the extent of the block. The third word specifies the first and last registers, in the order used in an LM instruction, containing such values. The implications of this method are as follows:

1. All stored values referencing a processor or user program must be kept in the specified contiguous locations, and these locations must contain only this type of value.

2. All values referencing the processor and the user area which are kept in registers must be kept in the registers specified by CSREG and PSREG. The registers specified by CSREG and PSREG must be in sequence and contain only this type of value.

Note: Register 0 is considered to contain absolute values and is never updated.

I/O PROCESSING

There are two types of I/O processing done by a processor or user program: terminal and disk.

Terminal I/O

The terminal I/O buffer (TMBUF) is at the end of the communications area. This buffer is used for all input or output operations with the terminal. Input from the terminal is requested by means of a call to the Executive. The input is placed at the beginning of this buffer and is terminated by a new line (NL) character. Only one line of input at a time can be requested.

A word in the communications area (BUFPTR) indicates the next available byte in the buffer when it is being used for output. As output is placed in the buffer, this pointer is updated. The Executive empties the buffer and resets the pointer to zero under the following conditions:

1. When it is full. This is indicated by means of a call to the Executive.

2. When input is requested.

3. When the program is swapped, except when the output inhibit flag (OPFLG) is ON. This flag is set during the time interval in the output routine when the output pointer does not correspond with the actual contents of the buffer.

4. On final exit.

The output inhibit flag allows the compiler to defer output during any time-slice interrupts. This is necessary when certain values, such as the terminal buffer pointer, are being changed.

Since the same buffer is used for input and output, it is not possible to place output data in the buffer unless all the input has been processed. The Executive assures that all output has been performed before a line of input is requested. The compiler (actually, the run-time library) considers it an error if more input is provided than is needed by the GET statement. If less data is provided than is required to satisfy the GET, additional lines are requested, one at a time, until the correct amount has been entered. To request input, a '?' character is typed out on the terminal.

Note: The Executive places an end-of-file (X'01') character in the byte indicated by the output pointer, so the last byte in the buffer must always be left empty. An end-of-file character placed in the buffer by the user program will be treated by the Executive as an end-of-file. Other undefined characters are reserved for use of the Executive terminal handling routines. The presence of these characters in the terminal I/O buffer may cause unpredictable results.

Disk I/O

The user is permitted up to four open disk files in his program. These files may be in any combination of input or output modes. Files may be closed and the same or new ones opened in either input or output mode. Each active file has an I/O buffer of 3712 bytes assigned to it. The first 3440 bytes are used as the I/O area to read and write one half track of data. The number of half tracks of disk space available per file is established by the user through the FILE command.

Disk input or output is accomplished by calling the Executive. The Executive is in no way concerned with the internal format of the data files. It performs the physical I/O in buffer-size blocks (3440 bytes) in the user area. Data files can be "reset" by special use of the SVC.

INTERRUPT HANDLING

During execution, a user can specify the actions to be performed if end-of-file or arithmetic interrupt conditions occur by use of ON statements. The runtime library determines the processing required by ON-conditions.

The compiler can control which arithmetic interrupts are in effect by use of the Set Program Mask (SPM) instruction. Upon entry to the compiler, the contents of the program mask are indeterminate.

When an arithmetic interrupt occurs, it is processed by the Executive as follows:

1. The program check old PSW is stored in an entry (PSW2SV) in the communications area (to be used by the processor). The PSW contents are updated when a program is swapped. The update is performed on the language processor's base address if SVC code 11 has not been given, and on the user program base address if SVC code 11 has been given.

2. Control is transferred to a routine that alters the base register and branches to the appropriate interrupt processing routine. During compilation, the displacement of the interrupt routine from the beginning of the compiler is found in location ARINTRP; during execution, the interrupt routine itself begins at location ARINTRP. When all processing for the interrupt is completed, and execution is to continue with the instruction following the interrupted instruction, the Executive is informed by the use of SVC code 7 or 8. To resume processing at a different point, the address portion of the saved PSW can be modified by the processor prior to issuing the SVC.

3. The Executive saves the contents of all registers at interrupt time in the user communications area. As indicated above, requests to the Executive are issued by loading a request code in register 0 and executing the SVC instruction in the communications area. Thus, the call to the Executive (requesting return to the interrupted code) causes execution of an SVC instruction. Since the SVC is an interrupt itself, the register save area contains the contents of the registers at the time the SVC was given, not the contents of the registers at the time of the arithmetic interrupt. Therefore, if registers are to be preserved, the language processors must save and restore all registers while processing an arithmetic interrupt.

An end-of-file condition is detected by a special return from the Executive from a read request. When this situation occurs, the runtime library branches directly to the library end-of-file routine.

SWAP-INHIBITED SITUATIONS

During the process of compilation, the contents of the registers cannot always correspond with the relocation specification in the communications area. This situation normally occurs twice:

1. When the program is being initiated.

2. When the program area is being changed from its compile-time configuration to its runtime configuration.

Each non-swap interval may have a maximum duration of 16 milliseconds. In 1 above, no special action must be taken as swapping will not take place during the first 16 milliseconds. In 2 above, the processor and the program can be made non-swappable (non-relocatable) by setting a word in the communications area to nonzero (SWPFLG). The swap flag may be set only once during a compilation.

END OF COMPILATION

When compilation is completed successfully, the compiler must perform at least three functions before relinquishing control to the user program. These functions are:

1. Determine whether there are one or more unused 2048-byte blocks at the top of the program area that are not needed for data files. If so, these must be returned by means of the SVC.

2. Adjust the processor and program relocation information to reflect the new situation. While this is being done, the program is made non-swappable.

3. Go to the Executive with an SVC code 11 to indicate that compilation is complete (return is to the first byte of the user's program (PRGBN); see SVC code 11 write-up).

When control is transferred to the user's program, the user program must:

1. Indicate that the program is now swappable by setting SWPFLG to zero.

2. Open all data files that can be opened at this time and specify how much additional memory will be required. During compilation, the language processor should keep track of the maximum number of data files that can be open at any one time and calculate the additional core required to hold these files. In addition, language processors should keep track of the files that are to be opened during the execution of the program and do a multiple OPEN at this time. The reason for combining these functions is that every call for an OPEN will result in a swap and every call for more memory will probably result in a swap. Therefore, if all four data files are used, the Executive can open all four files and obtain the additional memory required with one swap instead of five.

DETAILED FORMAT DESCRIPTIONS

The communications area of the user work area and specific portions of the user terminal table provide the basis for compiler/Executive interactions. The formats of these areas and of data file tables maintained for active data files are described below.

COMMUNICATIONS AREA

The names and sizes of various fields in the communications area, as well as their starting locations, are given below.

Location (Hex)	Name	Size In Words	Description
0	PRGBN	16	Initial entry point of compiled program.

Location (Hex)	Name	Size In Words	Description
40	CSPTR	1	Compiler save pointer. Contains displacement from beginning of program of start of block of values to be updated when processor swapped (relocated).
44	CSLTH	1	Compiler save length. Contains length in bytes (must be multiple of four) of area occupied by values to be updated when processor swapped.
48	CSREG	1	Compiler registers. First halfword contains first register used for values to be updated when compiler swapped, second halfword contains last such register.
4C	PSPTR	1	PSPTR through PSREG are used in the same way as CSPTR through CSREG for values to be updated when user program is swapped.
50	PSLTH	1	
54	PSREG	1	
58	BUFPTR	1	Buffer pointer. Contains displacement from beginning of terminal I/O buffer (TMBUF) to next available byte to be used for output.
5C	OPFLG	1	Output inhibit flag. Normally is 0. Set to 1 if output buffer should not be emptied when program swapped.
60	SWPFLG	1	Swap flag. Normally is 0. Set to 1 at the time when a program cannot be swapped.
64	ARINTRP	7	Location to which control is transferred on arithmetic interrupt.
80	PSW1SV	2	Save area for PSW when program or compiler swapped.
88	PSW2SV	2	Save area for PSW when arithmetic interrupt occurs (can be referenced by program).
90	PSW3SV	2	Same as PSW2SV, but used by the Executive when swapping only.
98	PSW4SV	2	Special PSW save area.
AO	BASPROC	1	Processor base address at swap time.
A4	BASUSER	1	User area base address at swap time.
A8	BUFLTH	1	Length of terminal I/O buffer.
AC	UTTLOC	1	Address of UTT (user terminal table) of this user.

Location (Hex)	Name	Size In Words	Description
BO	SAVREG	24	Locations in which registers are saved when program is swapped. Registers are saved in the order fixed-point 1 through 0 then floating-point 0 through 6.
110	DATE	1	Address of the location where the current date is maintained by the Executive in the form YYMMDD (6 bytes).
114	PDMPBGN	1	Displacement to beginning of area to be PDUMPed.
118	PDMPEND	1	Displacement to end of area to be PDUMPed (SVC code 13) or number of lines to dump (SVC code 14).
11C	FILEPTR	1	Pointer (displacement) to a table containing four logical records concerning data files.
120	NOERMSG	1/2	Number of error messages output to terminal during compilation.
122	FILENBR	1/4	Logical file number (file reference number) to be read/written.
123	FILE2K	1/4	Number of additional 2048-byte blocks of core required when an OPEN (SVC code 21) is issued.
124	SAVER0	1	Language processor saves register 0 here prior to issuing an SVC. The Executive restores register 0 from here prior to returning control.
128	STATTAB	1	Pointer to the statistical table.
12C	SVCINST	1/2	SVC instruction for execution by language processors.
12E		2 1/2	Locations reserved for additional communications cells.
138	USCCW	262	User program CCW's (used by the Executive).
550	TMBUF	Any length (between 256 and 5900 bytes, specified by BUFLTH)	Terminal I/O buffer

UTT DATA AVAILABLE TO LANGUAGE PROCESSOR

Fields of the user terminal table which can be read (but not modified) by the CALL/360-OS PL/I compiler are described below.

Name	Function	How Long (Bytes)	How Adjusted
L#LANG	Language processor name (coded value, 0-N)	1	BB
L#LADR	Address of this language processor's entry in the language processor table	4	WB
L#N2048	Number of 2048-byte blocks allocated	1	BB
L#SOURC	Length of source program (in bytes)	2	HWB
L#SADDR	Displacement from base of program area to beginning of source (in bytes)	4	WB
L#WIDTH	Line width (in characters)	1	BB
L#NLINE	Number of lines of source statements	2	HWB
L#FLG2	Expanded user program storage allocated (bit 6) Set when Reg 2 on SVC code 6 = 0 Not set when Reg 2 on SVC code 6 ≠ 0 Not set on initial entry (bit 6 can be addressed symbolically as L#ESBIT)	1	BB
L#FILE1D through L#FILE4D	Record number of data file link that was just read/written	1(ea.)	BB
L#FILE1E through L#FILE4E	Number of data file links in the file	1(ea.)	BB
L#FILE1F through L#FILE4F	Maximum number of permissible data file links	1(ea.)	BB
L#FILE1G through L#FILE4G	File type and state	1(ea.)	BB

WB = word boundary   HWB = halfword boundary   BB = byte boundary

DATA FILE TABLE

Data file tables (16 bytes long each) are maintained by the language processor and the Executive. Four of these tables may be active at any one time (although the design is such that the tables may be expanded).

The table addresses (displacements) are in a table that in turn is pointed to (displacement) by FILEPTR in the communications area. This addressing hierarchy is illustrated schematically by Figure F-2.

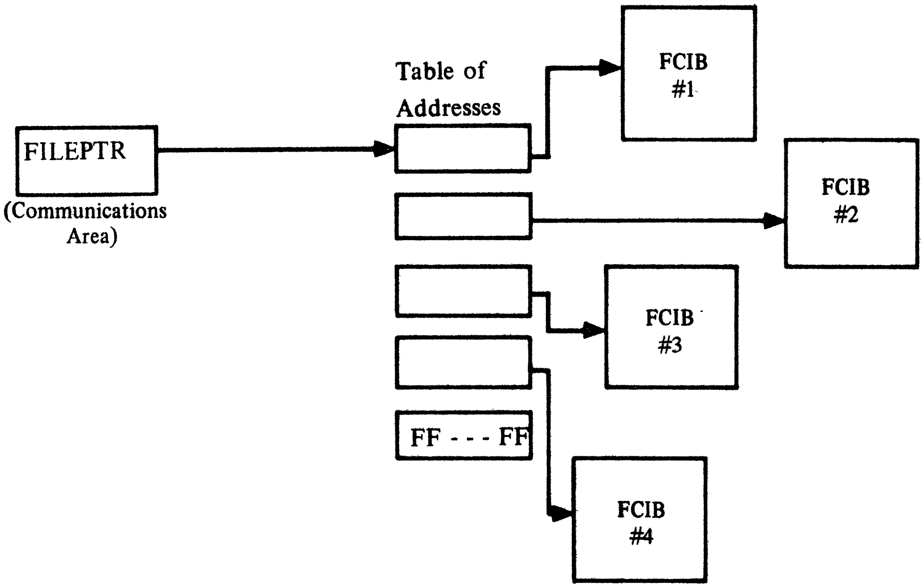


Figure F-2. Referencing Data File Tables

Notes:

1. The table of addresses must begin on a word boundary.

2. The table of addresses is terminated by a word of all 1's (binary 1111 = hexadecimal F).

3. The file tables must begin on word boundaries.

4. Entries in the table of addresses point to file tables in ascending numerical order, that is, first address is logical file #1, second address is logical file #2, etc.

5. If an entry in the table of addresses is 0, then the corresponding file table does not exist.

The format of each data file table is detailed below.

Byte	Contents
0-10	Filename (left-justified with blank padding). If this field is 0, then the file is not in use.
11	OPEN/mode flag bit 7 - set by the compiler if the file type is input bit 6 - set by the compiler if the file type is output (both bits 6 and 7 ON is legal) bit 5 - set by the Executive after the file has been opened bit 4 - set by compiler before Executive opens the file bit 3 - set by the compiler after OPEN status has been validated bits 2-1 - not used at present bit 0 - set by the Executive when the last link of the file has been read/written

Byte Contents

12 OPEN/I/O return code

A. Set by the Executive after an OPEN as follows:

0 = OPEN successfully done.
1 = OPEN not done because of an unrecoverable I/O error.
2 = OPEN not done because file does not exist.
3 = OPEN not done because file is locked (this code can only occur if bit 6 in byte 11 is ON).
4 = OPEN not done because file is already in use.
5 = Not used.
6 = Not used.
7 = OPEN not done because file is not a data file.
8 = Not used.
9 = Not used.

Byte Contents

B. Set by the Executive after an I/O as follows:

0 = READ/WRITE successfully done.
1 = Unrecoverable I/O error.
2 = READ/WRITE not done because user's data file space is exhausted.
3 = READ/WRITE not done because file mode (input or output) is incorrect.
4 = WRITE not done because no room in save storage.

Note: Bit 0 will be set ON if the last link was just read/written.

13 Record number of data file link that was just read/written (same as L#FILEnD).

14 Number of data file links in the file (same as L#FILEnE).

15 Maximum number of permissible data file links (same as L#FILEnF).

Before a language processor can open a file, the following actions must be performed:

1. Set the filename in bytes 0 through 10.

2. Set bits 4 and 6 and/or 7 in OPEN/mode flag ON and bit 5 OFF.

Upon return from the OPEN, the language processor will find bit 5 in the OPEN/mode byte set ON and a return code will be set in the OPEN/I/O return code byte. Language processors should not alter bit 5 in the OPEN/mode flag as this could cause erroneous opens to be performed.

OUTPUT BUFFER FORMAT

The output buffer (TMBUF) starts at a fixed increment from the beginning of the user program (and communications) area, (location hex 550). The Executive always supplies the EOF. When the Executive decides that a time quantum is up, it places the EOF character at the end of the present string of output (provided OPFLG is not set). An output buffer pointer is maintained by the user program so that the Executive knows where to place the EOF character. When control is returned after an SVC code 1, 2, or a time quantum swap, when OPFLG is zero, the Executive resets the pointer to zero.

FORMAT OF DATE INFORMATION

As noted under "Communications Area", the location of the area containing the current date is stored in the communications area, beginning in location hex 110. The format of the area containing the date is:

    YYMMDD

where:

    YY is last two digits of the year (69, 70, 71, etc.)
    MM is month (01-12)
    DD is day (01-31)

SUPERVISOR CALL (SVC) INSTRUCTION

The use of an SVC is one way that language processors have of communicating their needs to the Executive. In most cases, control is returned to the instruction following the SVC. Because CALL/360-OS uses so many SVC codes, the probability of conflict with other user-defined SVC's is very high. In order to minimize this conflict, CALL/360-OS uses one SVC and passes a code in register 0 indicating the kind of action required. Prior to loading register 0 with the SVC code, the language processor saves register 0 in the communications area at SAVER0. After the Executive gets control from the SVC, it moves SAVER0 into the proper place in SAVREG. The Executive then uses SAVER0 as a temporary working area. The SVC to be used is placed in SVCINST by the Executive. A language processor simply executes this location. In cases where parameters are passed back and forth (other than the SVC code), they are usually passed in register 2.

SVC code 0 -- Final exit. Control is not returned. Any terminal output that is in the output buffer is transmitted to the terminal.

SVC code 1 -- Output buffer full exit. Control is returned after the output buffer has been transmitted to the terminal. The Executive will place the EOF character in the buffer and reset the pointer to zero.

SVC code 2 -- Input from terminal required. The job is swapped out and placed in the new job queue. Control is returned after the output buffer has been transmitted to the terminal and an input line has been received from the terminal. It is required that the program RUN routines place a '?' at the end of the output buffer to indicate to the user that input is required. To ensure that this '?' is printed on the terminal only when the terminal is ready to read data, the Executive detects its presence, turns on the L#QMPT bit in the UTT, and places an EOF in the last position of the output buffer. If only a '?' is to be printed, L#EFBT is also turned on to eliminate the need for further processing of the buffer by M#LISO. M#ISRD will finally write the '?' on the terminal as part of the CCW chain which issues the read to the terminal in preparation for receiving the data input.

The input line is placed at the beginning of the output buffer and is terminated by a new line (NL) character. Only one input line at a time is allowed. The input line must be used or removed before the next input line is requested or the buffer used for output.

SVC code 3 -- Write to disk. Control is returned after the write has been completed. The address to begin writing (displacement) is placed in register 2 and the file reference number (1-4) to write is set in FILENBR in the communications area. Upon return, the file's record number will be incremented by one and the OPEN/I/O return code byte in the appropriate data file table will be set with a code as follows:

0 = write successfully completed
1 = unrecoverable I/O error
2 = write not done because user has filled his available space
3 = write not done because file type is input
4 = write not done because no room left in save storage

In addition to this code, bit 0 of byte 11 will be turned ON if the last permissible data file link was just written.

This SVC (and SVC code 4) operates in a special mode in order to implement the RESET (data file pointer) function. If register 2 is negative when the SVC (and SVC code 4) is issued, the Executive will not perform any I/O but will simply reset the user's internal pointers to the initial value of the data file disk address, reset the appropriate record number counter to zero, and return control to the caller immediately.

SVC code 4 -- Read from disk. Control is returned after the read has been completed. The address to begin reading into is in register 2 (displacement) and the file reference number (1-4) that is to be read is placed in FILENBR in the communications area. Upon return, L#FILEnD (n=1,2,3,or 4) and the appropriate counter in the user's data file table will be incremented by one and the OPEN/I/O return code byte in the appropriate data file table will be set with a code as follows:

0 = read successfully done
1 = unrecoverable I/O error
2 = no more data
3 = read not done because file type is output

In addition to this code, bit 0 of byte 11 will be turned ON if the last data file link was just read.

See special case for RESET described in SVC code 3 write-up.

SVC code 5 -- Memory give-back. Control is returned immediately. Register 2 contains the number of 2048-byte blocks that are being handed back. The Executive assumes that the memory being given back comes from the top (high address) of the user program area.

SVC code 6 -- Need more memory. This SVC operates in two modes. If the amount of memory is known, the number of 2048-byte blocks needed is specified in register 2. Under this condition, the Executive will allocate the required number of blocks and return control to the instruction following the SVC. The program will probably be swapped while waiting for memory to "free-up", and will be located at a different place in core when control is returned. The additional allocated memory is adjoined to the end of the original program area. When register 2 contains a nonzero value, the L#ESBIT bit of L#FLG2 in the UTT is not changed.

If the amount of additional memory is not known, register 2 is set to 0. In this event, the Executive will add an arbitrary amount of core, set L#ESBIT in the UTT ON and restart the job.

The Executive will permit more than one request for more memory. However, there will be an arbitrary upper limit (3-5) to the number of times this will be permitted. If this limit is exceeded, the program will be aborted. Optimally, processors should not seek additional memory more than once because this activity degrades system performance.

A halfword (NOERMSG) is provided in the communications area for language processors to save information pertaining to error messages output during compilation. When more core is requested and the amount is not known, the user should not see error messages repeated at his console. To avoid reissuing compilation error messages when the program is restarted, the Executive will save NOERMSG in the communications area upon an SVC code 6 exit. It will restore this halfword when the program is restarted. This halfword will be set to zero the first time the user calls for a compilation.

SVC code 7 -- This SVC code is the same as SVC code 8.

SVC code 8 -- Exit from arithmetic interrupt routine. Control is returned to the instruction following the instruction that caused the interrupt unless the compiler has modified the address portion of PSW2SV. In this case, control is returned to the specified location. Boundary limits are checked and L#ITPB is reset. PSW2SV is moved to PSW1SV, and control is returned by a relinquish.

SVC code 9 -- This SVC code is presently unassigned.

SVC code 10 -- This SVC code is a combination of codes 3 and 22. However, if the write is not successful, the close (code 22) is not performed.

SVC code 11 -- Compilation complete. This is used by the Executive to facilitate user program space management. It should be executed when an object program is about to be entered from the compiler. Return is to the first byte of the user program area (PRGBN) and is immediate. No register modification is performed by the Executive while processing the SVC. Thus, when the SVC is issued, all registers must contain the values assumed by the object program.

SVC code 12 - This is a debugging feature. It will cause the user's area to be dumped onto the printer, provided the printer is not presently being used for any other purpose (the printer I/O does not queue). No return is made after this SVC. This SVC is allowed only from the command console.

SVC code 13 -- Not used.

SVC code 14 -- Not used.

SVC code 15 -- Not used.

SVC code 16 -- Not used.

SVC code 17 -- End of CALL/360-OS PL/I phase 1 compilation.

SVC code 18 -- Not used.

SVC code 19 -- Not used.

SVC code 20 -- Not used.

SVC code 21 -- OPEN data file(s) and request additional memory. This SVC will cause each file whose OPEN/mode byte in the data file table has been properly set (bit 6 and/or 7 ON, bit 4 ON, and bit 5 OFF) to be opened and additional memory equal to the number of 2048-byte blocks specified in FILE2K in the communications area to be added onto the user's area. A return code will be provided for each file that was marked to be opened. FILE2K will be cleared by the Executive before returning control to the user program.

Logical file n can be closed and reopened under a different or the same name. The act of reopening the file informs the Executive that old file n is closed.

SVC code 22 -- Close data file n.

The logical file number (n) of the file to be closed must be contained in the communications area, byte FILENBR, before issuing SVC 22.

Upon return, byte 12 of the appropriate data file table will be set to 0 to indicate that the close was successfully done.

SVC code 23 -- Controlled abort. This SVC is issued when the compiler detects a condition that "should never happen" but did. An error code (range 0-999) is placed in register 2. The Executive will print this code on the communications console as a debugging aid for compiler writers. Control will not be returned after this SVC.

APPENDIX G - CALL/360-OS PL/I COMPILER MAINTENANCE

MODULE STORAGE

The compiler source and object code is kept on disk files in partitioned data sets. The names of the disk packs, data set names, and their usage are:

DISK PACKS:

External Label	Internal Label	System
OSSPLI	RTSLC2	OS/RTS

DATA SETS:

RTS1.PLI.SOURCE	Compiler source
RTS1.PLI.OBJECT	Compiler object
RTS1.PLI.MACLIB	Macro library

UPDATE AND ASSEMBLY

```
//UPDTE        JOB     PLI,RTS,MSGLEVEL=1
//             EXEC    PGM=IEBUPDTE
//SYSPRINT     DD      SYSOUT=A
//SYSUT1       DD      DSNAME=RTS1.PLI.SOURCE,DISP=OLD
//SYSUT2       DD      DSNAME=RTS1.PLI.SOURCE,DISP=OLD
//SYSIN        DD      *
./             CHANGE  NAME=member name

                       ***************************************
                       *    CARDS TO BE ADDED OR CHANGED     *
                       ***************************************

/*

//A            JOB     766,K,MSGLEVEL=1
//ASM          EXEC    PGM=IEUASM,REGION=50K,PARM=LOAD
//SYSLIB       DD      DSNAME=RTS1.PLI.MACLIB,DISP=OLD
//SYSUT1       DD      DISP=OLD,DSNAME=SYS1.UT1
//SYSUT2       DD      DISP=OLD,DSNAME=SYS1.UT2
//SYSUT3       DD      DISP=OLD,DSNAME=SYS1.UT3
//SYSPRINT     DD      SYSOUT=A
//SYSPUNCH     DD      DUMMY
//SYSIN        DD      DSNAME=RTS1.PLI.SOURCE(member name),DISP=OLD
//SYSGO        DD      DSNAME=RTS1.PLI.OBJECT(member name),DISP=OLD
/*
```

Notes:

1. The member names in the CALL/360-OS PL/I source data set and macro library are listed under "CALL/360-OS PL/I Member Names" below.

2. To update a member in the MACLIB, the JCL in the UPDATE step must be changed from RTS1.PLI.SOURCE to RTS1.PLI.MACLIB.

3. For update, cards in the deck that have the same sequence numbers as cards on the disk will replace the cards on the disk. Other cards will be inserted into the proper place. The cards in the deck must be in collating sequence.

4. Cards can be deleted from a member by placing the following card (in collating sequence) in the change deck:

   ```
   ./  DELETE SEQ1=nnnnnnnn,SEQ2=nnnnnnnn
   ```

   where SEQ1 is the first card to be deleted and SEQ2 is the last card to be deleted.

5. An entire member from the data set can be replaced by using the following card instead of the CHANGE card:

   ```
   ./  REPL  NAME=member name
   ```

6. When needed, the NUMBER card may be used to renumber the updated member. Place the NUMBER card after the CHANGE or REPL card.

   ```
   ./  NUMBER  SEQ1=ALL,NEW1=100,INCR=100
   ```

7. The ADD card may be used to add a new member. The JCL cards in the update procedure must be changed as follows:

   ```
   //  EXEC    PGM=IEBUPDTE,PARM=NEW
   ```

   Remove the SYSUT1 card and insert the following card.

   ```
   ./  ADD     NAME=member name
   ```

<u>LINK EDIT</u>

A load module can be created and saved on disk so that it can be executed. The link edit procedure is shown below for one phase of the compiler.

```
//MERTON    JOB      1600,'MERTON L,880',MSGLEVEL=1
//JOBLIB    DD       DSNAME=RTS.LOAD,DISP=(OLD,PASS),UNIT=2314,        X
//                   VOLUME=SER=ATPDO7
//LJED      EXEC     PGM=IEWL,PARM='XREF,LIST,LET,NCAL',REGION=96K
//SYSPRINT DD        SYSOUT=A
//SYSLIN    DD       DSNAME=SYSIN
//SYSLMOD   DD       DSNAME=RTS.LOAD(PH1),DISP=OLD,UNIT=2314,          X
//                   SPACE=(1024,(200,21,1),VOLUME=SER=ATPDO7
//SYSUT1    DD       UNIT=(SYSDA,SEP=(SYSLMOD,SYSLIN)),                X
//                   SPACE=(1024,(200,20))
//SYSLIB    DD       DSNAME=RTS1.PLI.OBJECT,DISP=OLD,UNIT=2314,        X
//                   VOLUME=SER=RTSLC2
//SYSIN     DD       *
            INCLUDE SYSLIB ($CCONT)

            ************************************
            *    INCLUDE CARD FOR EACH MEMBER  *
            ************************************
/*
```

<u>Note</u>: The load module for this link edit step will be stored in the data set RTS.LOAD (PH1) on disk pack ATPD07. The load module can be put on any disk pack under any data set name by changing the SYSLMOD control card.

CALL/360-OS PL/I MEMBER NAMES

The member names in the CALL/360-OS PL/I source data set and macro library are listed in two groups, "Compilation Member Names" and "Runtime Member Names."

COMPILATION MEMBER NAMES

The phase in which each routine is used is indicated by P1 for phase 1 (compilation phase), P2 for phase 2 (wrap-up phase), or P12 for both phases.

$ABAL	- P1	$DRET	- P1	$NCSDV	- P1
$ACGEN	- P1	$EDGN	- P1	$NCVT	- P1
$ANCRE	- P1	$ENDES	- P1	$NEXP	- P1
$APRC	- P1	$ENDON	- P1	$NLSIB	- P1
$AREXP	- P1	$EXPND	- P1	$NMULT	- P1
$ASIDX	- P12	$EYPND	- P1	$NOPCV	- P1
$ATKN	- P12	$FIND	- P12	$NOPRT	- P1
$BEGIN	- P1	$FLG	- P1	$NPRE	- P1
$BGET	- P1	$FMT	- P1	$OPEN	- P1
$BLPRC	- P1	$FNB	- P12	$OPMZO	- P1
$BONSA	- P1	$FORI	- P1	$SCDV	- P1
$BPUT	- P1	$FPDL	- P1	$SVC	- P12
$BRNH	- P1	$FSYM	- P1	$TCODE	- P1
$CALL	- P1	$FVAR	- P1	$TOPR	- P1
$CATEG	- P1	$GPUT	- P12	$TRIAD	- P1
$CCONT	- P1	$GTRIAD	- P1	$VASGA	- P1
$CERR	- P12	$HAINI	- P2	$VASGC	- P1
$CIF	- P1	$HCTP	- P2	$VCLR	- P1
$CNT	- P1	$HDVTP	- P2	$VDSAC	- P1
$CON	- P1	$HLNTP	- P2	$VGTMP	- P1
$CRVT	- P1	$HRTLL	- P2	$VINSA	- P1
$CSTOP	- P1	$HSCAL	- P2	$WBACK	- P1
$DCLGN	- P1	$HTCR	- P2	$WCONT	- P2
$DDS	- P2	$MCWU	- P2	$WCTCT	- P12
$DEXP	- P1	$NATTP	- P1	$WEXP	- P12
$DIOS	- P1	$NCALL	- P1	$WSTEP	- P12
$DOCS	- P1	$NCONS	- P1	$XERR	- P1
$DOG	- P1				

RUNTIME MEMBER NAMES

The runtime routines are all loaded in phase 2. However, none are executed until the runtime phase.

IHEABU	- P2	IHEIOD	- P2	IHESQW	- P2
IHEABW	- P2	IHEIOG	- P2	IHESQZ	- P2
IHEABZ	- P2	IHEIOP	- P2	IHESVC	- P2
IHEATL	- P2	IHEIOX	- P2	IHETHL	- P2
IHEATS	- P2	IHEJXI	- P2	IHETHS	- P2
IHEATW	- P2	IHELDI	- P2	IHETNL	- P2
IHEATZ	- P2	IHELDO	- P2	IHETNS	- P2
IHECLOSE	- P2	IHELNL	- P2	IHETNW	- P2
IHECSC	- P2	IHELNS	- P2	IHETNZ	- P2
IHECSM	- P2	IHELNW	- P2	IHEUPA	- P2
IHECSS	- P2	IHELNZ	- P2	IHEVCA	- P2
IHEDCN	- P2	IHEMXB	- P2	IHEVCS	- P2
IHEDDI	- P2	IHEMXL	- P2	IHEVFA	- P2
IHEDDO	- P2	IHEMXS	- P2	IHEVFB	- P2
IHEDDP	- P2	IHEMZU	- P2	IHEVFC	- P2
IHEDIA	- P2	IHEMZW	- P2	IHEVFD	- P2
IHEDIB	- P2	IHEMZZ	- P2	IHEVFE	- P2
IHEDIM	- P2	IHEONREV	- P2	IHEVPA	- P2
IHEDIO	- P2	IHEOPEN	- P2	IHEVPB	- P2

IHEDMA	- P2
IHEDNC	- P2
IHEDOA	- P2
IHEDOB	- P2
IHEDOM	- P2
IHEDUM	- P2
IHEDZW	- P2
IHEDZZ	- P2
IHEEFL	- P2
IHEEFS	- P2
IHEERN	- P2
IHEERR	- P2
IHEEXL	- P2
IHEEXS	- P2
IHEEXW	- P2
IHEEXZ	- P2
IHEGPUT	- P2
IHEHTL	- P2
IHEHTS	- P2
IHEIOA	- P2
IHEIOB	- P2
IHEPDF	- P2
IHEPDL	- P2
IHEPDS	- P2
IHEPDW	- P2
IHEPDX	- P2
IHEPDZ	- P2
IHERSET	- P2
IHESAD	- P2
IHESAF	- P2
IHESHL	- P2
IHESHS	- P2
IHESMF	- P2
IHESMG	- P2
IHESMH	- P2
IHESMX	- P2
IHESNL	- P2
IHESNS	- P2
IHESNW	- P2
IHESNZ	- P2
IHESQL	- P2
IHESQS	- P2
IHEVPC	- P2
IHEVPE	- P2
IHEVSC	- P2
IHEVTB	- P2
IHEXIB	- P2
IHEXIL	- P2
IHEXIS	- P2
IHEXIU	- P2
IHEXIW	- P2
IHEXIZ	- P2
IHEXXL	- P2
IHEXXS	- P2
IHEXXW	- P2
IHEXXZ	- P2
IHEYGF	- P2
IHEYGL	- P2
IHEYGS	- P2
IHEYGW	- P2
IHEYGX	- P2
IHEYGZ	- P2

APPENDIX H - DIAGNOSTIC MESSAGES

COMPILATION ERROR MESSAGES

Ident. Code	Error Message	Calling Routines
1	PROCEDURE STATEMENT SUPPLIED	$CNT
2	ILLEGAL '___' STATEMENT--NULL CLAUSE SUPPLIED	$CNT
3	'___' NOT STATEMENT TYPE, IGNORED	$CNT
4	NOT STATEMENT TYPE, ASSIGNMENT ASSUMED	$CNT
5	EXTRA ')', IGNORED	$ATKN
6	'||' NOT SUPPORTED--CHANGED TO '|'	$ATKN
7	IDENTIFIER TRUNCATED TO 8 CHARS	$ATKN
8	EXPONENT MISSING	$ATKN
9	CONSTANT NOT SUPPORTED--DECIMAL USED	$ATKN
10	DELIMITER OR SEPARATOR MUST FOLLOW CONSTANT	$ATKN
11	BIT STRINGS NOT SUPPORTED--CHARACTER USED	$ATKN
12	'___' NOT SUPPORTED, BLANK ASSUMED	$ATKN
13	')' SUPPLIED BEFORE ';'	$ATKN
14	'___' ILLEGAL DELIMITER--IGNORED	$ATKN
15	ILLEGAL ASSIGNMENT STATEMENT	$ACGEN
16	ERROR AT '___'	$ABAL $ANCRE $APRC $BEGIN $BGET $BONSA $BPUT $BRNH $CALL $CIF $CON $CRVT $CLOSE $CSTOP $DCLGN $DCOS $DDS $DIOS $DOG $DRET $EDGN $FLG $FMT $FORI $ACGEN $OPEN
17	'___'NOT ENTRY NAME	$CALL
18	STRUCTURES NOT SUPPORTED--'___' IGNORED	$DCLGN
19	NO FORMAT ITEM FOR DATA	$FLG
20	ILLEGAL USE OF '___' ATTRIBUTE	$ANCRE
21	PREVIOUS DECLARATION OR USE OF '___'	$DCLGN
22	ILLEGAL 'DO' INDEX	$DOG

Ident. Code	Error Message	Calling Routines	
23	ARRAY EXPRESSION ILLEGAL	$AREXP	$NOPRT
24	MAXIMUM NO. OF BLOCKS EXCEEDED	$CNT	
25	ILLEGAL 'WHILE' CLAUSE	$DOG	
26	DUPLICATE '___' CLAUSE	$DOG	
27	'___' AFTER 'END' ILLEGAL--IGNORED	$EDGN	
28	UNDEFINED FORMAT	$EDGN	
29	ILLEGAL USE OF '___'	$EDGN $FLG	$NOPRT $NEXP
30	UNLABELED FORMAT STATEMENT	$FMT	
31	FILE NAME NOT INPUT FILE	$BGET	
32	'___' WHERE ',' EXPECTED--SKIPPING TO '___'	$APRC	
33	ILLEGAL USE OF '___' IN DATA INPUT LIST	$BGET	
34	'SKIP' OPTION ILLEGAL HERE	$BGET	
35	'___' NOT FILE NAME--IGNORED	No longer used	
36	'TO' MISSING AFTER 'GO'	$BRNH	
37	ILLEGAL STATEMENT LABEL--STATEMENT IGNORED	$BRNH	
38	ITERATIVE 'DO' REQUIRED	$DOG	
39	'THEN' CLAUSE MISSING--NULL ASSUMED	$CIF	
40	FILE NAME MISSING	$BONSA $CLOSE	$OPEN
41	'___' NOT FILE NAME	$BONSA $CALL	$CLOSE $OPEN
42	'EOF' ON 'SYSIN' USELESS	No longer used	
43	ILLEGAL FILE DESIGNATION	$BPUT $BGET	$CLOSE $OPEN
44	UNRECOGNIZABLE ON-CONDITION	$BONSA	
45	LABEL ILLEGAL HERE--IGNORED	$CON	
46	LABEL MISSING	$APRC	
47	ILLEGAL RETURNS ATTRIBUTES-- DEFAULT RETURNS ATTRIBUTES USED	$ANCRE	$APRC
48	ILLEGAL OPTION ON EXTERNAL PROCEDURE STATEMENT	$APRC	
49	FILENAME NOT OUTPUT FILE	$BPUT	
50	NON-PRINT FILE--'SKIP' OPTION ILLEGAL	$BPUT	

Ident. Code	Error Message	Calling Routines
51	RETURN STATEMENT ILLEGAL IN ON-UNIT	$DRET
52	RETURNS ATTRIBUTE ILLEGAL IN EXTERNAL PROCEDURE--SKIPPING TO ';'	$DRET
53	ILLEGAL RETURNS EXPRESSION	No longer used
54	FILE NAME MISSING--'SYSIN' ASSUMED	No longer used
55	'___' NOT FILE NAME--ON-CONDITION IGNORED	No longer used
56	MULTIPLE DECLARATION FOR '___'--THIS DECLARATION USED	$APRC $BLPRC
57	USE OF '___' HERE CONFLICTS WITH PREVIOUS USAGE	$ANCRE $NPRE $BLPRC $OPEN $CLOSE
58	ILLEGAL LIST AFTER ATTRIBUTE '___' FOR IDENTIFIER '___'	$ABAL
59	FOR IDENTIFIER '___' ATTRIBUTE '___' CONFLICTS WITH PREVIOUS ATTRIBUTES	$ABAL
60	LIST MISSING AFTER ATTRIBUTE '___'	$ABAL
61	'___' ILLEGAL ATTRIBUTE--IGNORED	No longer used
62	PRECISION ATTRIBUTE IS ILLEGAL--DEFAULT USED	$ANCRE
63	ILLEGAL PARAMETER ATTRIBUTES FOR '___'	$ANCRE
64	ILLEGAL SCALE FACTOR FOR '___'--IGNORED	$ANCRE $NOPRT
65	NOT ALL DIMENSION EXPRESSIONS ARE CONSTANTS	$ANCRE
66	FOR STRING '___'--LENGTH NOT A CONSTANT	$ANCRE
67	'___' HAS ILLEGAL LENGTH--255 USED	$ANCRE
68	'___' HAS ILLEGAL '*' DIMENSION OR STRING LENGTH	$ANCRE
69	ATTRIBUTE FOR FILE '___' CONFLICTS WITH PREVIOUS DECLARATION OR USE	$ANCRE
70	DUPLICATE '___' DESIGNATION--LAST USED	$DOCS $DIOS
71	LIST MISSING AFTER '___'	$BPUT $DIOS $DOCS
72	'___' ILLEGAL OPTION--SKIPPING TO '___'	No longer used
73	DATA AND FORMAT LIST MISSING	$DIOS
74	FORMAT LIST MISSING	$DIOS
75	ILLEGAL I/O EXPRESSION--SKIPPING TO '___'	$DDS
76	ILLEGAL DATA OUTPUT ITEM	$DDS
77	EXPRESSION ILLEGAL IN 'GET' DATA LIST	$DDS

Ident. Code	Error Message	Calling Routines
78	ITERATION FACTOR NOT PARENTHESIZED	$FLG
79	'___' ILLEGAL FORMAT ITEM--SKIPPING TO '___'	$FLG
80	'___' INCOMPLETE FORMAT ITEM--SKIPPING TO '___'	$FLG
81	ILLEGAL COMPLEX FORMAT ITEM--SKIPPING TO '___'	$FLG
82	'___' NOT FORMAT LABEL--SKIPPING TO '___'	$FLG
83	FORMAT ITEM HAS INCORRECT NO. OF FIELDS	$FORI
84	CONVERSION ERROR--SCALE FACTOR TOO LARGE	$TCODE
85	PROGRAM INCOMPLETE--REQUIRED 'END' STATEMENTS SUPPLIED	$ASIDX
86	'___' WHERE OPERAND EXPECTED	$NEXP
87	'___' WHERE OPERATOR EXPECTED	$NEXP
88	ARGUMENT LIST MISSING FROM SUBPROGRAM CALL	No longer used
89	ILLEGAL OPERAND FOR COMPARISON OPERATOR	$NOPRT
90	BIT STRINGS NOT SUPPORTED FOR COMPARISON OPERATORS	No longer used
91	OPERAND OF '___' MUST BE BIT STRING	$NOPRT
92	PREFIX OPERATORS NOT SUPPORTED FOR BIT STRINGS	No longer used
93	ILLEGAL ASSIGNMENT	$NOPRT
94	INCORRECT NO. OF ARGUMENTS IN SUBPROGRAM	$NOPRT $NPRE
95	INVALID ARGUMENT ATTRIBUTE IN SUBPROGRAM CALL	$NOPRT
96	INCORRECT NO. OF SUBSCRIPTS FOR ARRAY	$NEXP $NOPRT
97	EMPTY DECLARATION--IGNORED	No longer used
98	CONVERSION OF CONSTANT PRODUCES EXPONENT OUT OF RANGE	$NCVT
99	CONSTANT VALUE OR PRECISION TOO LARGE	$NCVT
100	COMPILER ERROR	$NEXP $TCODE $NOPRT
101	EOF MISSING	No longer used
102	PROGRAM TOO LARGE	No longer used
103	INCOMPLETE COMMENT OR CHARACTER STRING CONSTANT	$ATKN $FNB
104	IMPROPER ARRAY BOUND	$ANCRE
105	OPERAND INCOMPATIBLE WITH REQUIRED ATTRIBUTES	$NOPCV $NOPRT
106	LINE NUMBER NOT FOLLOWED BY BLANK	$ASIDX $ATKN

Ident.		
Code	Error Message	Calling Routines
107	SOURCE STMTS AFTER END OF PROGRAM IGNORED	$EDGN
108	SEVERE DIAGNOSTICS, EXECUTION PREVENTED	$EDGN
109	ILLEGAL STATEMENT	No longer used
110	ILLEGAL TITLE DESIGNATION	$CLOSE $OPEN
111	'___' ILLEGAL OPERATOR	$NEXP
112	IMPROPER RELATIONAL EXPRESSION	$NEXP
113	SPACE FOR COMPILED CODE EXCEEDED	$HAINI
114	(iixxxxxx) PROGRAM ERROR - COMPILE TERMINATED	$CERR
115	STRING TOO LONG -- FIRST 255 USED	$ATKN

EXECUTION ERROR MESSAGES

Ident.		
Code	Error Message	Calling Routines
002	ERROR	No longer used
024	PRINT OPTION FORMAT ITEM FOR NON-PRINT FILE	IHEIOX
025	EXTRA INPUT DATA IGNORED	IHEIOA IHEDDI
123	ILLEGAL FILENAME	IHERSET IHECLOSE
124	NOT OPENED	IHEDDI IHERSET
125	UNRECOVERABLE I/O ERROR	IHEIOA IHEOPEN IHEIOG
126	DOES NOT EXIST	IHEOPEN
127	LOCKED	IHEOPEN
128	IN USE	IHEOPEN
129	PROTECTED	No longer used
130	NOT A DATA FILE	IHEOPEN
131	A SHARED FILE	No longer used
132	NOT INPUT TYPE	No longer used
133	NOT OUTPUT TYPE	No longer used
134	ILLEGAL ATTRIBUTES	IHEOPEN
135	EXCEEDS FOUR FILES OPEN	IHEDUM IHEOPEN
136	*DIRECTORY MISSING	IHEOPEN IHECLOSE
140	END OF FILE ENCOUNTERED	IHEIOG IHELDI
200	X LT 0 IN SQRT(X)	IHESQL IHESQS

Ident. Code	Error Message	Calling Routines
201	X GR 174.6 IN EXP(X)	IHEEXL IHEEXS
202	X LT OR = 0 IN LOG(X) OR LOG2(X) OR LOG 10(X)	IHELNL IHELNW IHELNS IHELNZ
203	ABS(X) GE (2**50)*K IN SIN(X) OR COS(X) (K=PI)	IHEEXZ IHESNZ IHESNL
204	ABS(X) GE (2**50)*K IN TAN(X) (K=PI)	IHETNL IHETNZ
205	X TOO NEAR SINGULARITY AND WILL GIVE OVERFLOW	No longer used
206	X=Y=0 IN ATAN(Y,X)	IHEATL IHEATS
207	ABS(X) GT 174.6 IN SINH(X) OR COSH(X)	No longer used
208	ABS(X) GE 1 IN ATANH(X)	IHEHTL IHEHTS
209	X=0, Y LE 0 IN X**Y	IHEXIL IHEXIW IHEXIS IHEXIZ
210	X=0, Y NOT POSITIVE REAL IN X**Y	IHEXXW IHEXXZ
211	Z=+I OR -I IN ATAN(Z) OR Z=+1 OR -1 IN ATANH(Z)	IHEATW IHEATZ
212	ABS(X) GE (2**18)*K IN SIN(X) OR COS(X) (K=PI)	IHEEXW IHESNW IHESNS
213	ABS(X) GE (2**18)*K IN TAN(X) (K=PI)	IHETNS IHETNW
300	OVERFLOW	IHEABZ IHESNW IHEDZW IHESNZ IHEDZZ IHESQW IHEEXW IHESQZ IHEEXZ IHETNL IHEMZW IHETNS IHEMZZ IHETNW IHEPDF IHETNZ IHEPDL IHEXIL IHEPDS IHEXIS IHEPDW IHEXIW IHEPDX IHEXIZ IHEPDZ IHEYGF IHESHL IHEYGL IHESHS IHEYGS IHESMF IHEYGW IHESMG IHEYGX IHESMH IHEYGZ IHESMX
301	INEXPLICABLE I/O ERROR	IHEDUM IHECLOSE IHEIOD IHERSET IHELDI
320	FIXEDOVERFLOW	IHEABU IHEMZU IHEABW
330	ZERODIVIDE	IHEDZW IHEMZU IHEDZZ

Ident. Code	Error Message	Calling Routines	
340	UNDERFLOW	IHEDZW	IHESMH
		IHEDZZ	IHESMX
		IHEMZW	IHEXIL
		IHEMZZ	IHEXIS
		IHEPDF	IHEXIW
		IHEPDL	IHEXIZ
		IHEPDS	IHEYGF
		IHEPDW	IHEYGL
		IHEPDX	IHEYGS
		IHEPDZ	IHEYGW
		IHESMF	IHEYGX
		IHESMG	IHEYGZ
500	SUBSCRIPT RANGE	IHEDDI	
600	CONVERSION	IHEVPE	
601	CONVERSION ERROR IN F-FORMAT	No longer used	
602	CONVERSION ERROR IN E-FORMAT	IHEVPC	
604	ERROR IN CONVERSION FROM CHARACTER STRING TO ARITHMETIC	IHELDI	IHEDCN
605	ERROR IN CONVERSION FROM ARITHMETIC TO CHARACTER STRING	IHELDI	
606	ERROR IN CONVERSION FROM FIXED TO FLOAT	IHELDI	
607	ERROR IN CONVERSION FROM FLOAT TO FIXED	IHELDI	
700	INCORRECT E(W,D,S) SPECIFICATION	No longer used	
701	F FORMAT W SPECIFICATION TOO SMALL	No longer used	
702	A FORMAT W UNSPECIFIED AND LIST ITEM NOT TYPE STRING	IHEDOB	
704	A FORMAT W UNSPECIFIED ON INPUT	IHEDOB	
705	SUBSTRING NOT IN DATA AREA	IHELDO	
706	MAXIMUM STRING LENGTH EXCEEDED	IHEIOD	
802	END OF OUTPUT FILE	IHEIOD	IHERSET
803	IMPROPER NO. OF SUBSCRIPTS FOR DATA INPUT VARIABLE	IHEDDI	
805	DATA NAME NOT FOUND IN SYMBOL TABLE	IHEDDI	
806	SUBSCRIPT NOT IN USER AREA	IHEDDI	
807	RECURSIVE BLOCK OR ON-UNIT	IHESAD	
808	DATA I/O ON INTERNAL FILE	IHEDDI	
809	ILLEGAL LABEL VARIABLE GO TO	IHESAF	
810	EDIT I/O ON INTERNAL FILE	IHEDIO	

Ident. Code	Error Message	Calling Routines
811	DECLARED ENVIRONMENT NOT COMPATIBLE WITH INPUT FILE	IHEIOA
902	PROGRAM ERROR - EXECUTION TERMINATED	IHEERR

APPENDIX I - MAXIMUM SIZE OF SOURCE PROGRAM

The maximum size of a CALL/360-OS PL/I source program is determined by storage requirements at various stages of the compilation and execution processes. The values stated in this appendix apply to operation of the CALL/360-OS system on a System/360 computer having 512K bytes of main storage. These values are subject to change and should be regarded accordingly.

STORAGE REQUIRED AT INPUT OF PROGRAM

Maximum area allocated to hold source statements of a program to be compiled under CALL/360-OS provides for 28,848 source characters or 800 source lines. The effective maximum is determined by whichever limit is reached first. Either limit permits approximately 800 CALL/360-OS PL/I statements at 30 characters per statement.

STORAGE REQUIRED TO COMPILE PROGRAM

The size of this area is determined by six items. Three of them are fixed in size and three are variable. They are listed below.

Fixed

Communications area	1,360 bytes
Terminal I/O buffer	3,000 bytes
Compiler fixed-size working storage	14,960 bytes
TOTAL	19,320 bytes

Variable

Source program (28,848 byte max.)
Object program
Compiler variable-size working storage

The size of storage up to the maximum provided to the CALL/360-OS PL/I compiler (112K bytes) is determined by the following formula:

Size of area = 39,000 + 4*(bytes of source program)

STORAGE REQUIRED TO EXECUTE PROGRAM

The size of this area is determined by eight items. Two of them are fixed and six are variable. They are as listed below.

Fixed

Communications area	1,360 bytes
Terminal I/O buffer	3,000 bytes
TOTAL	4,360 bytes

Variable

Object program size
Static and constants storage } (62K byte max.)
Address constant area (1.6K byte min.)
Runtime library (7K min. & 60K max.)
Disk I/O buffers (multiples of 3,440 bytes) (max. of 4)
Dynamic storage (array and string storage)

The maximum size of this area as allowed by the CALL/360-OS system is 112K bytes.

EXAMPLES

Three examples to illustrate storage requirements follow.

EXAMPLE 1

A source program containing 290 source statements requires main storage locations as shown below.

1. Input storage used - 8176 bytes

2. Compiler area required -

39,000 + 4(8176)	=	71,704
	=	73,728 bytes allocated

3. Execution area used -

Communications area	=	1,360
Terminal I/O buffer area	=	3,000
Object program size	=	15,056
Static and constants area	=	1,176
Address constant area	=	1,752
Runtime library	=	23,792
Disk I/O buffer area	=	0
Dynamic storage	=	0
TOTAL	=	46,136 bytes

EXAMPLE 2

A program containing 15 source statements has the following requirements for main storage.

1. Input storage used - 344 bytes

2. Compiler area required -

39,000 + 4(344)	=	40,376	
	=	40,960	bytes allocated

3. Execution area used -

Communications area	=	1,360	
Terminal I/O buffer area	=	3,000	
Object program size	=	632	
Static and constants area	=	600	
Address constant area	=	1,780	
Runtime library	=	16,572	
Disk I/O buffer area	=	0	
Dynamic storage	=	0	
TOTAL	=	23,944	bytes

EXAMPLE 3

A program containing 434 source statements requires main storage as follows.

1. Input storage used - 18,287 bytes

2. Compiler area required -

39,000 + 4(18,287)	=	112,148	
	=	112,640	bytes allocated

3. Execution area used -

Communications area	=	1,360	
Terminal I/O buffer area	=	3,000	
Object program size	=	27,056	
Static and constants area	=	3,124	
Address constant area	=	3,000	
Runtime library	=	26,516	
Disk I/O buffer area	=	0	
Dynamic storage	=	0	
TOTAL	=	64,056	bytes

CALL/360-OS PL/I COMPILER SUBROUTINES

Under CALL/360-OS PL/I naming conventions, subroutines of the CALL/360-OS PL/I compiler are named $xxxxx, where xxxxx is a mnemonically suggestive symbol of functions performed. These subroutines are discussed in functional groups in Section 3, Volume I, of this manual. They are listed below in alphabetic order according to their mnemonics. If more than one entry point exists for a routine, multiple entry points are noted in the leftmost column. The next leftmost column shows the mnemonic commonly used in general discussion of the routine (for example, in this manual). A brief statement of function and the chart number for each routine are provided.

Note: The CALL/360-OS PL/I compiler subroutine names follow the naming convention stated above. However, there are some exceptions in the member names assigned to certain routines when stored in CALL/360-OS PL/I libraries. For the reader's convenience, the exceptions are noted in Figure J-4, which is a cross reference of compilation module calls to other compilation modules.

Entry Name	Routine Name	Function	Manual Location	Chart
$ABAL	$ABAL	Prepare bit mask and list of pointers for explicitly declared attributes for an identifier	I-135	27
$ACGEN	$ACGEN	Prepare an assignment statement for analysis	I-77	9
$ANCRE	$ANCRE	Translate attribute table ($ABTBL) into dictionary attribute node	I-138	28
$APRC	$APRC	Analyze syntax, create parameter declarations, and generate triads for internal procedure	I-78	10
$APRC2	$APRC	Analyze syntax, create parameter declarations, and generate triads for external procedure	I-78	10
$AREXP	$AREXP	Generate error message for illegal array expression	I-419	70
$ASIDX	$ASIDX	Advance scan index to next character in source stream	I-56	5
$ATKN	$ATKN	Create token table entries for syntactic units in source stream	I-57	6

Entry Name	Routine Name	Function	Manual Location	Chart
$ATKN2	$ATKN	See above. Used for initial entry only	I-57	6
$BGET	$BGET	Analyze a GET statement	I-168	33
$BEGIN	$BEGIN	Check syntax of BEGIN statement and generate part of prologue	I-80	11
$BLPRC	$BLPRC	Define the address of a statement label or entry name	I-141	29
$BONSA	$BONSA	Check legality of ON-condition and identify it	I-81	12
$BPUT	$BPUT	Analyze a PUT statement	I-170	34
$BRNH	$BRNH	Analyze a GO TO statement	I-82	13
$BRNH2	$BRNH	Analyze a GOTO statement	I-82	13
$CALL	$CALL	Analyze the syntax of a CALL statement	I-84	14
$CATEG	$CATEG	Determine whether next statement in token table is assignment statement	I-39	1
$CCONT	$CCONT	Initialize every area required for compilation	I-40	3
$CERR	$CERR	Signal hardware interrupt or unrecoverable error caused by the compiler	I-420	71
$CIF	$CIF	Analyze the syntax of an IF statement	I-85	15
$CNT	$CNT	Direct entokening of statements and determine which statement processor is required	I-42	2
$CON	$CON	Analyze syntax of ON statement and generate code to establish ON-condition address	I-86	16
$CRVT	$CRVT	Analyze a REVERT statement	I-88	17
$CSTOP	$CSTOP	Analyze the syntax of a STOP statement	I-89	18
$DCLGN	$DCLGN	Direct analysis and encoding of attributes for identifiers in a DCL statement and construct dictionary attribute list	I-142	30

Entry Name	Routine Name	Function	Manual Location	Chart
$DDS	$DDS	Direct generation of triads required for a data list	I-172	35
$DEXP	$DEXP	Build the triads required for an iterative DO-loop	I-90	19
$DIOS	$DIOS	Find file, skip, and data specification pointers of GET or PUT statement	I-174	36
$DOCS	$DOCS	Find file, title, and input/output attribute sections of OPEN or CLOSE statement	I-175	37
$DOG	$DOG	Check syntax of and generate triads for DO statement and create entry for program structure table	I-92	20
$DOG2	$DOG	As above, except for DO specification in I/O list	I-92	20
$DRET	$DRET	Analyze the syntax of a RETURN statement	I-94	21
$EDGN	$EDGN	Generate triads and perform housekeeping for closings associated with END statement	I-95	22
$EDGN2	$EDGN	As above, except only closes END statement generated by $EYPND to complete an iterative DO-loop	I-95	22
$ENDES	$ENDES	Perform processing required at end of ELSE unit	I-97	23
$ENDON	$ENDON	Perform processing required at end of on-unit	I-98	24
$EXPND	$EXPND	Determine dimensionality of an array expression, generate required DO statements, and build temporary variables for indices	I-99	25
$EYPND	$EYPND	Generate END statements to complete DO-loops of $EXPND	I-100	26
$FIND	$FIND	Search dictionary name list for name entry for an identifier; if none, create one	I-60	7
$FLG	$FLG	Perform syntax analysis and code generation for a format list	I-176	38
$FMT	$FMT	Direct translation of a FORMAT statement	I-179	39

Entry Name	Routine Name	Function	Manual Location	Chart
$FNB	$FNB	Advance scan index to next nonblank character in source stream	I-62	8
$FORI	$FORI	Create FED for expression in a format specification	I-180	40
$FORI2	$FORI	As above, but for constant only	I-180	40
$FPDL	$FPDL	Process format list for edit-directed I/O	I-181	41
$FSYM	$FSYM	Find definition for identifier in higher-numbered block	I-144	31
$FVAR	$FVAR	Find definition of variable; if none, create one	I-145	32
$GPUT	$GPUT	Edit 120-character line and place in terminal buffer	I-421	72
$GTRIAD	$GTRIAD	Get next available triad from triad table	I-422	73
$HAINI	$HAINI	Convert adcons to true addresses and reset user area relocation constants	I-392	62
$HCTP	$HCTP	Process constant table	I-394	63
$HDVTP	$HDVTP	Set pointers of dope vectors of all static arrays and strings	I-395	64
$HLNTP	$HLNTP	Process line number table	I-396	65
$HRTLL	$HRTLL	Determine and load required library routines; allocate fixed and address-modifiable library work space	I-397	66
$HSCAL	$HSCAL	Process initialization table and dope vector list to initialize constants and adcon areas	I-399	67
$HTCR	$HTCR	Collapse C, D, and I tables and J list during wrap-up	I-401	68
$MCWU	$MCWU	Perform housekeeping to prepare code and start execution	I-403	69

Entry Name	Routine Name	Function	Manual Location	Chart
$NATTP	$NATTP	Obtain attributes of argument of CALL or function reference	I-209	43
$NCALL	$NCALL	Generate triads to call a function or subprogram	I-210	44
$NCONS	$NCONS	Convert constant; search constant table for similar entry; if none, create one	I-423	74
$NCON	$NCONS	As above, except $NXFLG indicates the type of conversion required	I-423	74
$NCSDV	$NCSDV	Process subscript list of an array cross-section and begin construction of dope vector	I-212	45
$NCVT	$NCVT	Convert arithmetic source constant to arithmetic target type	I-425	75
$NEXP	$NEXP	Control generation of triads to evaluate expressions, assignment statement, and CALL statement entry name and argument list	I-214	46
$NLSIB	$NLSIB	Provide pointer to adcon for a library entry name	I-426	76
$NMULT	$NMULT	Generate triad to multiply subscript value by dimension multiplier of current array dimension	I-218	47
$NOPCV	$NOPCV	Convert operand to required type	I-219	48
$NOPRT	$NOPRT	Process all operators in operator stack whose priorities are greater than or equal to the current operator	I-222	49
$NPRE	$NPRE	Process top entry of operand stack	I-225	50
$OPEN	$OPEN	Analyze an OPEN statement	I-182	42
$CLOSE	$OPEN	Analyze a CLOSE statement	I-182	42
$OPMZO	$OPMZO	Determine the effective signs of triad operands and arrange them to optimize referencing	I-270	52

Entry Name	Routine Name	Function	Manual Location	Chart
$SCDV	$SCDV	Construct initialization table entry for dope vector	I-271	53
$SVC	$SVC	Interface with the CALL/360-OS system	I-452	82
$TCODE	$TCODE	Generate symbolic instructions from entries in a triad table	I-272	54
$TOPR	$TOPR	Process operands of current triad	I-279	55
$TRIAD	$TRIAD	Generate a single entry in triad table by referring to $NLOPN, $NROPN, and top entry of operator stack	I-226	51
$STRD	$TRIAD	Same as above but using contents of registers G0, P3, and P4	I-226	51
$VASGA	$VASGA	Select a register for assignment from adcon register portion of register table	I-280	56
$VASGC	$VASGC	Select register or pair of registers for assignment from computational register portion of register table	I-281	57
$VFREE	$VASGC	Determine whether a designated register can be freed	I-281	57
$VRSYN	$VASGC	Remove synonyms from a designated register table entry	I-281	57
$VSAVE	$VASGC	Store contents of designated register into temporary storage	I-281	57
$VCLR	$VCLR	Allocate space for and initialize register table	I-284	58
$VDSAC	$VDSAC	Convert compiler's representation of a storage address to a machine address	I-285	59
$VGTMP	$VGTMP	Allocate temporary storage in dynamic storage area for a block	I-288	60

Entry Name	Routine Name	Function	Manual Location	Chart
$VINSA	$VINSA	Generate machine-language instructions (object code) from symbolic instructions of $TCODE	I-290	61
$WBACK	$WBACK	Step from one segment of a table to the preceding segment	I-428	77
$WCONT	$WCONT	Initiate second (wrap-up) phase of the CALL/360-OS PL/I compiler	I-44	4
$WCTCT	$WCTCT	Release segment of a table to free pool	I-428	78
$WEXP	$WEXP	Add segment to a table and adjust pointers to it	I-428	79
$WSTEP	$WSTEP	Step from one segment of a table to the preceding segment	I-428	80
$XERR	$XERR	Construct, parameterize, and print diagnostic message	I-431	81

CALL/360-OS PL/I RUNTIME LIBRARY

Routines of the CALL/360-OS PL/I compiler library provide interface and computational services. Under CALL/360-OS PL/I naming conventions, library module and entry names begin with the prefix "IHE". Module (routine) names are composed of these three letters and three additional unique letters that identify the specific routines. An additional unique letter is appended to identify an entry point in the module.

The CALL/360-OS PL/I runtime library routines are discussed in functional groupings, corresponding to recognized packages of the library, in Section 5, Volumes II and III, of this manual. They are listed below in alphabetic order according to their mnemonics. A brief summary of the function of each routine is provided.

Note: The CALL/360-OS PL/I library runtime routines follow the naming conventions stated above and applied in this manual. However, there are some exceptions in the member names assigned to certain routines when stored in CALL/360-OS PL/I system libraries. For the reader's convenience, those exceptions are noted below:

IOB, IOD, IOP, IOX, and LDO (which are referred to in documentation as IHEIOB, IHEIOD, IHEIOP, IHEIOX, and IHELDO).

Routine Name	Function	Package	Manual Location
IHEABU	Binary Fixed Complex ABS - Calculate ABS(z) = SQRT(x**2 + y**2) where z = x + yI and x and y are binary fixed real numbers.	AFUNC	II-118
IHEABW	Short Float Complex ABS - Calculate ABS(z) = SQRT(x**2 + y**2) where z = x + yI and x and y are short floating-point real numbers.	AFUNC	II-120
IHEABZ	Long Float Complex ABS - Calculate ABS(z) = SQRT(x**2 + y**2) where z = x + yI and x and y are long floating-point real numbers.	AFUNC	II-122
IHEATL	Long Float Real Arctan - Calculate arctan(x) or arctan(y/x) where x is a long floating-point real number expressed in radians.	MFUNC	III-9
IHEATS	Short Float Real Arctan - Calculate arctan(x) or arctan(y/x) where x is a short floating-point real number expressed in radians.	MFUNC	III-5
IHEATW	Short Float Complex Arctan/Hyperbolic Arctan - Calculate arctan(z) or hyperbolic arctan(z) where z is a short floating-point complex expression.	MFUNC	III-16
IHEATZ	Long Float Complex Arctan/Hyperbolic Arctan - Calculate arctan(z) or hyperbolic arctan(z) where z is a long floating-point complex expression.	MFUNC	III-19
IHECLOSE	Close - Close a disk file.	IOMP	II-40
IHECSC	Character String Compare - Compare two character strings and return condition code.	SIMP	II-111
IHECSM	Character String Assignment - Assign a character string to a fixed-length target.	SIMP	II-113
IHECSS	Character String SUBSTR - Produce an SDV describing the SUBSTR pseudo-variable and function of a character string.	SIMP	II-115
IHEDCN	Character String to Arithmetic - Convert a fixed-length character string containing arithmetic constant or complex expression to an arithmetic target with specified scale, mode, and precision.	TCP	II-88
IHEDDI	Data-Directed Input - Handle data-directed input operations.	IOMP	II-41

Routine Name	Function	Package	Manual Location
IHEDDO	Data-Directed Output - Handle data-directed output, performing any necessary conversion operations.	IOMP	II-43
IHEDDP	Perform Calculation of the Subscript Values for an Array Element - Calculate subscript values for an array element using FCB and ADV.	IOMP	II-45
IHEDIA	F/E Format Input Director - Convert F/E-format external data to an internal data type.	TCP	II-76
IHEDIB	A-Format Input Director - Convert A-format external data to an internal data type during edit I/O.	TCP	II-78
IHEDIM	C-Format Input Director - Convert C-format external data to an internal C-format representation during edit I/O.	TCP	II-80
IHEDIO	Edit I/O Director - Interpret format code and direct control to required library routine.	IOMP	II-47
IHEDMA	Arithmetic Conversion Director - Set up intermodular flow required to convert data from one arithmetic data type to another.	TCP	II-96
IHEDNC	Arithmetic to Character String - Convert an arithmetic source with specified scale, mode, and precision to a character string.	TCP	II-90
IHEDOA	F/E-Format Output Director - Convert an internal data representation to an external F/E-format during edit I/O.	TCP	II-82
IHEDOB	A-Format Output Director - Convert an internal data representation to an external A-format during edit I/O.	TCP	II-84
IHEDOM	C-Format Output Director - Convert an internal data representation to an external C-format during edit I/O.	TCP	II-86
IHEDUM	Program Termination - Terminate current program, closing all disk files.	HIP	II-63
IHEDZW	Short Float Complex Division - Calculate z1/z2 in floating-point when z1=a+bI and z2=c+dI, and a,b,c, and d are short floating-point real numbers.	AFUNC	II-130

Routine Name	Function	Package	Manual Location
IHEDZZ	Long Float Complex Division - Calculate z1/z2 in floating-point when z1=a+bI and z2=c+dI, and a,b,c, and d are long floating-point real numbers.	AFUNC	II-132
IHEEFL	Long Float Real Error Function - Compute the error function of x or the complement of this function, where x is a long floating-point real expression.	MFUNC	III-25
IHEEFS	Short Float Real Error Function - Compute the error function of x or the complement of this function, where x is a short floating-point real expression.	MFUNC	III-22
IHEERN	Table of Error Messages and Indicators - Provide action codes of execution errors and runtime error messages.	HIP	II-64
IHEERR	Error Routine - Identify error condition and determine required action.	HIP	II-65
IHEEXL	Long Float Real EXP - Compute e**x where x is a long floating-point real expression.	MFUNC	III-30
IHEEXS	Short Float Real EXP - Compute e**x where x is a short floating-point real expression.	MFUNC	III-28
IHEEXW	Short Float Complex EXP - Compute e**x where x is a short floating-point complex expression.	MFUNC	III-32
IHEEXZ	Long Float Complex EXP - Compute e**x where x is a long floating-point complex expression.	MFUNC	III-34
IHEGPUT	Output Director - Place 120-character line in terminal I/O buffer.	MOPP	II-70
IHEHTL	Long Float Real Hyperbolic Arctan - Calculate hyperbolic arctan(x) where x is a long floating-point real expression.	MFUNC	III-14
IHEHTS	Short Float Real Hyperbolic Arctan - Calculate hyperbolic arctan(x) where x is a short floating-point real expression.	MFUNC	III-12
IHEIOA	List- or Edit-Directed GET Initiation and Termination - Initiate or terminate list- or edit-directed GET statement.	IOMP	II-48

Routine Name	Function	Package	Manual Location
IHEIOB	Output Initialization with or without Skipping - Initialize PUT statement with or without SKIP option.	IOMP	II-49
IHEIOD	Output Data to the Buffer Area and Communication with CALL/360-OS - Place converted data in buffer and request an SVC to Executive when buffer is filled.	IOMP	II-50
IHEIOG	Get Data Field from Input Buffer - Collect data from an input buffer.	IOMP	II-52
IHEIOP	Perform SKIP(w) Function for SYSPRINT - Perform the SKIP function for output print file.	IOMP	II-53
IHEIOX	Edited Horizontal Control Format Item - On input, space over next w characters. On output, for control format item, insert w blanks; for COLUMN(w), insert blanks up to w-th character.	IOMP	II-54
IHEJXI	Interleaved Array Indexer - Provide the byte address of the next element of an array.	AMP	III-89
IHELDI	List- and Data-Directed Input - Scan one item or the constant part of an assignment and assign it to internal variable.	IOMP	II-56
IHELDO	List-Directed Output - Handle list-directed output.	IOMP	II-58
IHELNL	Long Float Real Log - Calculate log(x) to the base e, base 2, or base 10 where x is a long floating-point real expression.	MFUNC	III-39
IHELNS	Short Float Real Log - Calculate log(x) to the base e, base 2, or base 10 where x is a short floating-point real expression.	MFUNC	III-36
IHELNW	Short Float Complex Log - Calculate the principal value of the natural log of z where z is a short floating-point complex expression.	MFUNC	III-42
IHELNZ	Long Float Complex Log - Calculate the principal value of the natural log of z where z is a long floating-point complex expression.	MFUNC	III-44
IHEMXB	Real Binary Fixed MAX/MIN - Find the maximum or minimum of a group of real fixed-point binary numbers.	AFUNC	II-124

Routine Name	Function	Package	Manual Location
IHEMXL	Real Long Float MAX/MIN - Find the maximum or minimum of a group of long floating-point real numbers.	AFUNC	II-128
IHEMXS	Real Short Float MAX/MIN - Find the maximum or minimum of a group of short floating-point real numbers.	AFUNC	II-126
IHEMZU	Binary Fixed Complex Mult/Div - Calculate z1*z2 or z1/z2, where z1 and z2 are fixed-point binary complex numbers.	AFUNC	II-134
IHEMZW	Short Float Complex Mult - Calculate z1*z2 in floating-point, when z1=a+bI and z2=c+dI and a,b,c, and d are short floating-point real numbers.	AFUNC	II-136
IHEMZZ	Long Float Complex Mult - Calculate z1*z2 in floating-point, when z1=a+bI and z2=c+dI and a,b,c, and d are long floating-point real numbers.	AFUNC	II-137
IHEONREV	On-ENDFILE and REVERT Initializer - Initialize the on-ENDFILE condition unit to the current unit.	HIP	II-68
IHEOPEN	Open - Open a disk file.	IOMP	II-59
IHEPDF	PROD-Interleaved Real Fixed Array - Equate a long or short floating-point real target to the product of all elements of an interleaved array of fixed-point real expressions.	AMP	III-91
IHEPDL	PROD-Interleaved Real Long Float Array - Equate a long floating-point real target to the product of all elements of an interleaved array of long floating-point real expressions.	AMP	III-95
IHEPDS	PROD-Interleaved Real Short Float Array - Equate a short floating-point real target to the product of all elements of an interleaved array of short floating-point real expressions.	AMP	III-93
IHEPDW	PROD-Interleaved Complex Short Float Array - Equate a short floating-point complex target to the product of all elements of an interleaved array of short floating-point complex expressions.	AMP	III-99

Routine Name	Function	Package	Manual Location
IHEPDX	PROD-Interleaved Complex Fixed Array - Equate a long or short floating-point complex target to the product of all elements of an interleaved array of fixed-point complex expressions.	AMP	III-97
IHEPDZ	PROD-Interleaved Complex Long Float Array - Equate a long floating-point complex target to the product of all elements of an interleaved array of long floating-point complex expressions.	AMP	III-101
IHERSET	Reset Disk Files - For output, write current half-track and reset disk and current buffer pointers; for input, reset disk and current buffer pointers.	IOMP	II-60
IHESAD	Initial Prologue, Expand DSA, End Prologue, Object Program Initiation - Provide DSA for block, obtain automatic storage for declared elements, and determine space required for object program.	MOPP	II-71
IHESAF	GO TO Interpreter - Update current DSA address if necessary and free chain elements up to the DSA to which the specified label belongs.	MOPP	II-73
IHESHL	Long Float Real Hyperbolic Sin/Cos - Calculate hyperbolic sin(x) or hyperbolic cos(x), where x is a long floating-point real expression.	MFUNC	III-54
IHESHS	Short Float Real Hyperbolic Sin/Cos - Calculate hyperbolic sin(x) or hyperbolic cos(x), where x is a short floating-point real expression.	MFUNC	III-52
IHESMF	SUM-Interleaved Real Fixed Array - Equate a long or short floating-point real target to the sum of all elements of an interleaved array of fixed-point real expressions.	AMP	III-103
IHESMG	SUM-Interleaved Real/Complex Short Float Array - Equate a short floating-point real or complex target to the sum of all elements of an interleaved array of short floating-point real or complex expressions, respectively.	AMP	III-105
IHESMH	SUM-Interleaved Real/Complex Long Float Array - Equate a long floating-point real or complex target to the sum of all elements of an interleaved array of long floating-point real or complex expressions respectively.	AMP	III-107

Routine Name	Function	Package	Manual Location
IHESMX	SUM-Interleaved Complex Fixed Array - Equate a long or short floating-point complex target to the sum of all elements of an interleaved array of fixed-point complex expressions.	AMP	III-109
IHESNL	Long Float Real Sin/Cos - Compute sin(x) or cos(x) where x is a long floating-point real expression in radians.	MFUNC	III-49
IHESNS	Short Float Real Sin/Cos - Compute sin(x) or cos(x) where x is a short floating-point real expression in radians.	MFUNC	III-46
IHESNW	Short Float Complex Sin/Cos - Calculate hyperbolic sine, hyperbolic cosine, sine, or cosine of an argument z, where z is a short floating-point complex expression.	MFUNC	III-57
IHRSNZ	Long Float Complex Sin/Cos - Calculate hyperbolic sine, hyperbolic cosine, sine, or cosine of an argument z, where z is a long floating-point complex expression.	MFUNC	III-60
IHESQL	Long Float Real SQRT - Compute SQRT(x) where x is a long floating-point real expression.	MFUNC	III-66
IHESQS	Short Float Real SQRT - Compute SQRT(x) where x is a short floating-point real expression.	MFUNC	III-63
IHESQW	Short Float Complex SQRT - Compute the principal value of the square root of z where z is a short floating-point complex expression.	MFUNC	III-68
IHESQZ	Long Float Complex SQRT - Compute the principal value of the square root of z where z is a long floating-point complex expression.	MFUNC	III-70
IHESVC	Library SVC Director - Interface with the CALL/360-OS system.	MOPP	II-74
IHETHL	Long Float Real Hyperbolic Tan - Calculate hyperbolic tan(x) where x is a long floating-point real expression.	MFUNC	III-80
IHETHS	Short Float Real Hyperbolic Tan - Calculate hyperbolic tan(x) where x is a short floating-point real expression.	MFUNC	III-78

Routine Name	Function	Package	Manual Location
IHETNL	Long Float Real Tan - Calculate tan(x) where x is a long floating-point real number expressed in radians.	MFUNC	III-75
IHETNS	Short Float Real Tan - Calculate tan(x) where x is a short floating-point real number expressed in radians.	MFUNC	III-72
IHETNW	Short Float Complex Tan/Hyperbolic Tan - Calculate tan(z) or hyperbolic tan(z), where z is a short floating-point complex expression.	MFUNC	III-82
IHETNZ	Long Float Complex Tan/Hyperbolic Tan - Calculate tan(z) or hyperbolic tan(z) where z is a long floating-point complex expression.	MFUNC	III-84
IHEUPA	Zero Real or Imaginary Part - Zero real part of complex arithmetic data and move pointer from real to imaginary part or zero imaginary part and/or get address of imaginary part.	TCP	II-92
IHEVCA	Data Analysis Routine - Create a DED to describe the scale, mode, and precision of a character representation of an arithmetic value.	TCP	II-109
IHEVCS	Complex External to String Director - Direct conversion of character representation of complex data to internal string data.	TCP	II-93
IHEVFA	Float Intermediate to Packed Decimal Intermediate - Direct conversion of floating-point intermediate to packed decimal intermediate.	TCP	II-99
IHEVFB	Float Intermediate to Fixed Binary - Direct conversion of floating-point intermediate number to fixed-point binary.	TCP	II-100
IHEVFC	Float Intermediate to Float Short or Long - Move a floating-point intermediate number into a floating-point short or long target data item.	TCP	II-101
IHEVFD	Fixed Binary to Float Intermediate - Direct conversion of a fixed-point binary source to a floating-point intermediate number.	TCP	II-102
IHEVFE	Float Source to Float Intermediate - Move a short or long floating-point binary number into LCA to make it available for use as a floating-point intermediate number.	TCP	II-103

Routine Name	Function	Package	Manual Location
IHEVPA	Packed Decimal Intermediate to Float Intermediate - Convert a packed decimal intermediate number to a long floating-point intermediate number and store in LCA.	TCP	II-104
IHEVPB	Packed Decimal Intermediate to F-Format - Convert a packed decimal intermediate number to an F-format character string and store in target-string data item.	TCP	II-105
IHEVPC	Packed Decimal Intermediate to E-Format - Convert a packed decimal intermediate number to an E-format character string and store in target-string data item.	TCP	II-106
IHEVPE	String with Format to Packed Decimal Intermediate - Convert a character string paired with an F/E-format element to packed decimal intermediate and store in LCA.	TCP	II-107
IHEVSC	Character String to Character String - Assign a fixed- or variable-length character string to a fixed- or variable-length character string.	TCP	II-95
IHEVTB	Table of Powers of Ten - Table of long-precision floating-point numbers representing powers of ten from 1 to 70.	TCP	II-108
IHEXIB	Real Fixed Binary Integer EXP - Calculate x**n, where x is a real fixed-point binary number and n is a positive integer.	AFUNC	II-138
IHEXIL	Real Long Float Integer EXP - Calculate x**n, where x is a long floating-point real number and n is an integer between -2**31 and 2**31 - 1.	AFUNC	II-142
IHEXIS	Real Short Float Integer EXP - Calculate x**n, where x is a short floating-point real number and n is an integer between -2**31 and 2**31 - 1.	AFUNC	II-140
IHEXIU	Z**N, Z Fixed Binary Complex - Calculate z**n, where z is a complex fixed-point binary number and n is a positive integer less than 2**31.	AFUNC	II-144
IHEXIW	Z**N, Z Short Float Complex - Calculate z**n, where z is a short floating-point complex number and n is an integer between -2**31 and 2**31 - 1.	AFUNC	II-146

Routine Name	Function	Package	Manual Location
IHEXIZ	Z**N, Z Long Float Complex - Calculate z**n, where z is a long floating-point complex number and n is an integer between -2**31 and 2**31 - 1.	AFUNC	II-148
IHEXXL	Long Float Real General EXP - Calculate x**y, where x and y are long floating-point real numbers.	AFUNC	II-152
IHEXXS	Short Float Real General EXP - Calculate x**y, where x and y are short floating-point real numbers.	AFUNC	II-150
IHEXXW	Short Float Complex General EXP - Calculate z1**z2 where z1 and z2 are short floating-point complex numbers.	AFUNC	II-154
IHEXXZ	Long Float Complex General EXP - Calculate z1**z2 where z1 and z2 are long floating-point complex numbers.	AFUNC	II-156
IHEYGF	POLY(A,X) (A and X Real Fixed) - For vector X, calculate: $A(m) + \sum_{j=1}^{n-m} A(m+j) * \prod_{i=0}^{j-1} X(p+i)$ For scalar X, calculate: $\sum_{j=0}^{n-m} A(m+j)*X**j$	AMP	III-111
IHEYGL	POLY(A,X) (A and X Real Long Float) - For vector X, calculate: $A(m) + \sum_{j=1}^{n-m} A(m+j) * \prod_{i=0}^{j-1} X(p+i)$ For scalar X, calculate: $\sum_{j=0}^{n-m} A(m+j)*X**j$	AMP	III-117
IHEYGS	POLY(A,X) (A and X Real Short Float) - For vector X, calculate: $A(m) + \sum_{j=1}^{n-m} A(m+j) * \prod_{i=0}^{j-1} X(p+i)$	AMP	III-114

Routine Name	Function	Package	Manual Location
	For scalar X, calculate: $\sum_{j=0}^{n-m} A(m + j)*X**j$		
IHEYGW	POLY(A,X) (A and X Complex Short Float) - For vector X, calculate: $A(m) + \sum_{j=1}^{n-m} A(m + j)*\prod_{i=0}^{j-1} X(p + i)$ For scalar X, calculate: $\sum_{j=0}^{n-m} A(m + j)*X**j$	AMP	III-123
IHEYGX	POLY(A,X) (A and X Complex Fixed) - For vector X, calculate: $A(m) + \sum_{j=1}^{n-m} A(m + j)*\prod_{i=0}^{j-1} X(p + i)$ For scalar X, calculate: $\sum_{j=0}^{n-m} A(m + j)*X**j$	AMP	III-120
IHEYGZ	POLY(A,X) (A and X Complex Long Float) - For vector X, calculate: $A(m) + \sum_{j=1}^{n-m} A(m + j)*\prod_{i=0}^{j-1} X(p + i)$ For scalar X, calculate: $\sum_{j=0}^{n-m} A(m + j)*X**j$	AMP	III-126

MACRO-MACRO CROSS REFERENCE

CALL/360-OS PL/I compiler support macros are described in Appendix C of this manual. Runtime support macros are discussed in Appendix D. Some of the macros in each group call other macros to perform required functions. Figure J-1 provides a cross reference between a macro and other macros called by that macro. (Refer to the named appendices for details.)

Called Macros

Calling Macros

	RTSSVC	RCON	IHEZAP	IHESYM	IHESCV	IHELWS	IHELIB	IHELBE	IHEFCB	IHEEXT	IHEERRCD	IHEDIF	IHECAL	IHEBRA	IHEADC	GTRD	GETKN	GENER	GCURR	CALRTS	CALL
CALL																					
CALLERR													●								
CALRTS	●																				
CKFCB											●									●	
CSVC																					
DED																					
DNODE																					
EXPG																			●		●
FAREA																					●
FIB																●					
GCURR																					
GENER																					●
GETKN																					
GFRST																					
GNEXT																					●
GNODE																					●
GPREV																					●
GTRD																					●
IHEADC																					
IHEBRA																					
IHEBXT																					
IHECAL																					
IHEDCV					●																
IHEDIF																					
IHEERRCD																					
IHEEXT																					
IHEFCB																					
IHEFCIB																					
IHEFROM																					
IHEHDR																					
IHEIOD										●		●		●							
IHELBE																					
IHELIB				●		●															
IHEMOPP																					
IHENAME																					
IHEOPENT									●		●										
IHEPCH																					
IHEPRV	●																				
IHERET																					
IHERST																					
IHESAV																					
IHESCV																					
IHESDR																					
IHESYM															●						
IHETLR																					
IHEZAP																					
INODE																					
INSI																					
LIBDEF			●				●	●													
MNODE																					
RCON																					
READDISK											●									●	
READTERM																				●	
RFIB																●					
RTSSVC																					
SETDISK																					
SETERRCD																					
SETFLCA																					
SETSDV																					
SKPTK																	●				
SYMDEF		●																			
TALLY																			●		
TGENER																		●			

Figure J-1. Macro-Macro Cross Reference

MODULE-MACRO CROSS REFERENCE

CALL/360-OS PL/I compiler modules are discussed in detail in Section 3 of this manual. Runtime library modules are described in Section 5. Many of these modules call CALL/360-OS PL/I macros to perform required functions. Figure J-2 provides a cross reference between a compiler module and macros called by that module. Figure J-3 provides a cross reference between a runtime library module and macros called by that module. (Refer to the named sections for details.)

Calling Modules

	$FSYM	$FPDL	$FORI	$FNB	$FMT	$FLG	$FIND	$EYPND	$EXPND	$ENDON	$ENDES	$EDGN	$DRET	$DOG	$DOCS	$DIOS	$DEXP	$DDS	$DCLGN	$CSTOP	$CRVT	$CON	$CNT	$CIF	$CERR	$CCONT	$CATEG	$CALL	$BRNH	$BPUT	$BONSA	$BLPRC	$BGET	$BEGIN	$ATKN	$ASIDX	$AREXP	$APRC	$ANCRE	$ACGEN	$ABAL	
CALL	●	●	●		●	●		●	●		●	●		●			●	●	●		●	●	●		●	●	●	●	●	●	●	●	●	●	●			●	●	●		CALL
CALLERR																																										CALLERR
CALRTS																																										CALRTS
CKFCB																																										CKFCB
CSVC												●													●	●									●							CSVC
DED																		●																								DED
DNODE												●							●													●										DNODE
EXPG			●			●							●					●						●				●	●	●										●		EXPG
FAREA								●																		●									●				●			FAREA
FIB		●															●	●				●	●	●									●					●				FIB
GCURR		●			●	●			●	●	●	●		●			●	●	●			●	●	●									●							●		GCURR
GENER			●		●	●						●	●	●	●	●		●	●	●	●	●	●	●				●	●	●	●	●	●	●			●	●		●	●	GENER
GETKN		●	●		●	●			●			●	●	●	●	●		●				●	●	●			●	●	●	●	●		●					●	●	●	●	GETKN
GFRST									●																																	GFRST
GNEXT																																										GNEXT
GNODE			●			●	●		●			●		●			●					●	●	●		●			●			●		●	●	●		●	●			GNODE
GPREV		●				●			●	●	●	●	●				●	●						●																		GPREV
GTRD		●	●		●	●			●	●	●	●	●	●			●				●	●	●					●	●	●		●	●	●				●				GTRD
IHEADC																																										IHEADC
IHEBRA																																										IHEBRA
IHEBXT																																										IHEBXT
IHECAL																																										IHECAL
IHEDCV																																										IHEDCV
IHEDIF																																										IHEDIF
IHEERRCD																																										IHEERRCD
IHEEXT																																										IHEEXT
IHEFCB																																										IHEFCB
IHEFCIB																																										IHEFCIB
IHEFROM																																										IHEFROM
IHEHDR																																										IHEHDR
IHEIOD																																										IHEIOD
IHELBE																																										IHELBE
IHELIB																																										IHELIB
IHEMOPP																																										IHEMOPP
IHENAME																																										IHENAME
IHEOPENT																																										IHEOPENT
IHEPCH	●	●	●	●	●	●	●	●	●		●	●	●		●	●	●	●	●	●	●	●		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●		●	●	IHEPCH
IHEPRV																																										IHEPRV
IHERET																																										IHERET
IHERST																																										IHERST
IHESAV																																										IHESAV
IHESCV																																										IHESCV
IHESDR																																										IHESDR
IHESYM																																										IHESYM
IHETLR																																										IHETLR
IHEZAP																				●																						IHEZAP
INODE																			●							●			●			●						●				INODE
INST																																										INST
LIBDEF																																										LIBDEF
MNODE										●	●	●	●																													MNODE
RCON																																										RCON
READDISK																																										READDISK
READTERM																																										READTERM
RFIB		●			●					●	●	●					●	●					●										●									RFIB
RTSSVC																																										RTSSVC
SETDISK																																										SETDISK
SETERRCD																																										SETERRCD
SETFLCA																																										SETFLCA
SETSDV																																										SETSDV
SKPTK			●		●	●			●			●	●	●				●	●	●	●	●						●	●	●	●		●	●				●	●	●	●	SKPTK
SYMDEF	●	●	●	●	●	●	●	●	●		●	●	●	●	●	●		●	●	●	●	●	●	●	●		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	SYMDEF
TALLY		●	●		●							●										●										●	●	●					●			TALLY
TGENER		●		●																		●													●	●						TGENER
	$FSYM	$FPDL	$FORI	$FNB	$FMT	$FLG	$FIND	$EYPND	$EXPND	$ENDON	$ENDES	$EDGN	$DRET	$DOG	$DOCS	$DIOS	$DEXP	$DDS	$DCLGN	$CSTOP	$CRVT	$CON	$CNT	$CIF	$CERR	$CCONT	$CATEG	$CALL	$BRNH	$BPUT	$BONSA	$BLPRC	$BGET	$BEGIN	$ATKN	$ASIDX	$AREXP	$APRC,	$ANCRE	$ACGEN	$ABAL	

Called Macros

Figure J-2. Compilation Module-Macro Cross Reference (Page 1 of 2)

Calling Modules

	$XERR	$WSTEP	$WEXP	$WCTCT	$WCONT	$WBACK	$VINSA	$VGTMP	$VDSAC	$VCLR	$VASGC	$VASGA	$TRIAD	$TOPR	$TCODE	$SVC	$SCDV	$OPMZO	$OPEN	$NPRE	$NOPRT	$NOPCV	$NMULT	$NLSIB	$NEXP	$NCVT	$NCSDV	$NCONS	$NCALL	$NATTP	$MCWU	$HTCR	$HSCAL	$HRTLL	$HLNTP	$HDVTP	$HCTP	$HAINI	$GTRIAD	$GPUT	$FVAR	
CALL	●				●		●		●		●		●	●	●				●	●	●	●	●		●		●	●	●		●			●				●				CALL
CALLERR																																										CALLERR
CALRTS	(TXTX)																																									CALRTS
CKFCB																																										CKFCB
CSVC					●										●																●			●						●		CSVC
DED							●								●													●										●				DED
DNODE																																										DNODE
EXPG	(TXT)																																									EXPG
FAREA															●				●						●																	FAREA
FIB																			●																							FIB
GCURR									●						●										●		●															GCURR
GENER															●				●	●	●	●			●	●			●													GENER
GETKN					●														●						●		●														●	GETKN
GFRST															●										●								●		●		●					GFRST
GNEXT															●										●								●		●		●					GNEXT
GNODE								●					●		●		●		●		●	●			●	●	●	●													●	GNODE
GPREV							●	●	●						●					●		●			●		●															GPREV
GTRD																			●																							GTRD
IHEADC																																										IHEADC
IHEBRA																																										IHEBRA
IHEBXT																																										IHEBXT
IHECAL																																										IHECAL
IHEDCV																																										IHEDCV
IHEDIF																																										IHEDIF
IHEERRCD																																										IHEERRCD
IHEEXT																																										IHEEXT
IHEFCB																																										IHEFCB
IHEFCIB																																										IHEFCIB
IHEFROM																																										IHEFROM
IHEHDR																																										IHEHDR
IHEIOD																																										IHEIOD
IHELBE																																										IHELBE
IHELIB																																										IHELIB
IHEMOPP																																										IHEMOPP
IHENAME																																										IHENAME
IHEOPENT																																										IHEOPENT
IHEPCH								●	●	●	●	●	●	●	●	●	●	●	●	●		●	●	●	●		●		●	●	●	●	●	●	●	●	●			●	●	IHEPCH
IHEPRV																																										IHEPRV
IHERET																																										IHERET
IHERST																																										IHERST
IHESAV																																										IHESAV
IHESCV																																										IHESCV
IHESDR																																										IHESDR
IHESYM																																										IHESYM
IHETLR																																										IHETLR
IHEZAP																																										IHEZAP
INODE																																									●	INODE
INST															●																											INST
LIBDEF																																										LIBDEF
MNODE							●	●	●						●										●																	MNODE
RCON																																										RCON
READDISK																																										READDISK
READTERM																																										READTERM
RFIB																			●																							RFIB
RTSSVC																																										RTSSVC
SETDISK																																										SETDISK
SETERRCD																																										SETERRCD
SETFLCA																																										SETFLCA
SETSDV																																										SETSDV
SKPTK																			●															●								SKPTK
SYMDEF	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●		●	●	●	●	●	●	●	●	●	●	●	●		●	●	SYMDEF
TALLY							●		●		●				●				●									●			●										●	TALLY
TGENER																																										TGENER

Called Macros

Figure J-2. Compilation Module-Macro Cross Reference (Page 2 of 2)

Calling Modules

	IHEIOA	IHEHTS	IHEHTL	IHEGPUT	IHEEXZ	IHEEXW	IHEEXS	IHEEXL	IHEERR	IHEERN	IHEEFS	IHEEFL	IHEDZZ	IHEDZW	IHEDUM	IHEDNC	IHEDOM	IHEDOB	IHEDOA	IHEDMA	IHEDIO	IHEDIM	IHEDIB	IHEDIA	IHEDDP	IHEDDO	IHEDDI	IHEDCN	IHECSS	IHECSM	IHECSC	IHECLOSE	IHEATZ	IHEATW	IHEATS	IHEATL	IHEADMP	IHEABZ	IHEABW	IHEABU	
CALL																																									CALL
CALLERR																																									CALLERR
CALRTS									●																																CALRTS
CKFCB	●																								(CONVDEC)																CKFCB
CSVC																																									CSVC
DED																																									DED
DNODE																																									DNODE
EXPG																																									EXPG
FAREA																																									FAREA
FIB																																									FIB
GCURR																																									GCURR
GENER																																									GENER
GETKN																																									GETKN
GFRST																																									GFRST
GNEXT																																									GNEXT
GNODE																																									GNODE
GPREV																																									GPREV
GTRD																																									GTRD
IHEADC																																									IHEADC
IHEBRA			●		●	●					●	●				●	●	●	●			●	●	●				●					●	●							IHEBRA
IHEBXT			●		●	●					●	●																					●	●				●	●		IHEBXT
IHECAL																									●	●	●														IHECAL
IHEDCV		●	●		●	●					●	●					●	●	●			●	●					●					●	●	●	●		●	●		IHEDCV
IHEDIF																●	●	●	●	●								●													IHEDIF
IHEERRCD	●	●	●						●						●			●			●					●	●	●		●		●	●		●	●					IHEERRCD
IHEEXT																●	●	●	●			●	●	●				●													IHEEXT
IHEFCB	●																																								IHEFCB
IHEFCIB	●																				●						●														IHEFCIB
IHEFROM	●																										●														IHEFROM
IHEHDR		●	●	●	●	●	●	●			●	●	●	●		●	●		●	●	●	●	●	●				●	●	●	●		●	●	●	●		●	●	●	IHEHDR
IHEIOD																●		●	●			●	●	●																	IHEIOD
IHELBE																																									IHELBE
IHELIB		●	●	●	●	●	●	●			●	●	●	●		●	●	●	●	●		●	●	●				●	●	●	●		●	●	●	●	●	●	●	●	IHELIB
IHEMOPP									●																																IHEMOPP
IHENAME	●								●	●					●										●	●	●														IHENAME
IHEOPENT	●																										●														IHEOPENT
IHEPCH		●											●					●			●									●		●	●								IHEPCH
IHEPRV																																									IHEPRV
IHERET	●								●									●							●	●	●					●									IHERET
IHERST			●		●	●	●	●			●	●	●	●		●	●		●				●	●				●	●	●	●		●	●	●	●		●	●	●	IHERST
IHESAV		●	●		●	●	●	●			●	●	●	●		●	●	●	●	●		●	●	●				●	●		●		●	●	●	●		●	●	●	IHESAV
IHESCV							●	●																																	IHESCV
IHESDR	●								●																●	●	●					●									IHESDR
IHESYM																																									IHESYM
IHETLR		●	●	●	●	●	●	●			●	●	●	●		●	●	●	●	●	●	●	●	●				●	●	●	●	●	●	●	●	●		●	●	●	IHETLR
IHEZAP				●																																					IHEZAP
INODE																																									INODE
INST																																									INST
LIBDEF	●								●	●					●						●				●	●	●					●									LIBDEF
MNODE																																									MNODE
RCON																																									RCON
READDISK																																									READDISK
READTERM																																									READTERM
RFIB																																									RFIB
RTSSVC		●	●	●	●	●	●	●	●		●	●	●	●	●	●	●	●	●	●		●	●	●				●		●		●	●	●	●	●		●	●		RTSSVC
SETDISK																																									SETDISK
SETERRCD																																									SETERRCD
SETFLCA																										●															SETFLCA
SETSDV																									●	●	●														SETSDV
SKPTK																																									SKPTK
SYMDEF																																									SYMDEF
TALLY																																									TALLY
TGENER																																									TGENER

Called Macros

Figure J-3. Runtime Module-Macro Cross Reference (Page 1 of 3)

Calling Modules

Called Macros	IHESQL	IHESNZ	IHESNW	IHESNS	IHESNL	IHESMX	IHESMH	IHESMG	IHESMF	IHESHS	IHESHL	IHESAF	IHESAD	IHERSET	IHEPDZ	IHEPDX	IHEPDW	IHEPDS	IHEPDL	IHEPDF	IHEOPEN	IHEONREV	IHEMZZ	IHEMZW	IHEMZU	IHEMXS	IHEMXL	IHEMXB	IHELNZ	IHELNW	IHELNS	IHELNL	IHELDO	IHELDI	IHEJXI	IHEIOX	IHEIOP	IHEIOG	IHEIOD	IHEIOB
CALL																																								
CALLERR																																				●			●	
CALRTS													●																											
CKFCB																																		(CALLCP)						
CSVC																																								
DED																																								
DNODE																																							(INITIATE)	
EXPG																																								
FAREA																																								
FIB																																								
GCURR																																								
GENER																																								
GETKN																																								
GFRST																																								
GNEXT																																								
GNODE																																								
GPREV																																								
GTRD																																								
IHEADC																																								
IHEBRA		●	●			●	●	●	●	●	●				●	●	●	●	●	●									●											
IHEBXT		●	●							●	●																		●	●										
IHECAL																																	●	●		●	●			●
IHEDCV	●	●	●	●	●	●	●	●	●	●	●				●	●	●	●	●	●									●		●	●			●					
IHEDIF																																								
IHEERRCD	●			●	●							●	●	●							●	●									●	●	●	●		●	●		●	●
IHEEXT						●	●	●	●						●	●	●	●	●	●																				
IHEFCB														●																			●	●		●	●	●		
IHEFCIB																																	●	●		●	●			●
IHEFROM																																	●	●		●	●	●		●
IHEHDR	●	●	●	●	●	●	●	●	●	●	●			●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●				●	
IHEIOD																																								
IHELBE																																								
IHELIB	●	●	●	●	●	●	●		●	●	●				●	●	●	●	●	●			●	●	●	●	●	●	●	●	●	●			●					
IHEMOPP												●	●																											
IHENAME												●	●																							●	●	●		●
IHEOPENT																																								●
IHEPCH														●																			●	●						
IHEPRV																																								
IHERET												●	●	●							●	●											●	●			●	●	●	●
IHERST	●	●	●	●	●	●	●	●	●	●	●				●	●	●	●	●	●			●	●	●	●	●	●	●	●	●	●			●	●				
IHESAV	●	●	●	●	●	●	●	●	●	●	●				●	●	●	●	●	●			●	●	●	●	●	●	●	●	●	●			●					
IHESCV					●			●																	●															
IHESDR												●	●	●							●	●											●	●		●	●	●	●	●
IHESYM																																								
IHETLR	●	●	●	●	●	●	●	●	●	●	●			●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●				●	
IHEZAP																																								
INODE																																								
INST																																								
LIBDEF												●	●	●		●	●	●	●		●	●											●	●		●	●	●	●	●
MNODE																																								
RCON																																								
READDISK																																		●				●		
READTERM																																		●				●		
RFIB																																								
RTSSVC	●	●	●	●	●	●	●		●	●	●		●	●	●	●	●	●	●	●	●		●	●	●	●	●	●	●	●	●	●		●	●					
SETDISK																																								
SETERRCD																																								
SETFLCA																																	●			●				
SETSDV																																					●			
SKPTK																																		●						
SYMDEF																																								
TALLY																																								
TGENER																																								

Figure J-3. Runtime Module-Macro Cross Reference (Page 2 of 3)

Calling Modules

	IHEXGZ	IHEYGX	IHEYGW	IHEYGS	IHEYGL	IHEYGF	IHEXXZ	IHEXXW	IHEXXS	IHEXXL	IHEXIZ	IHEXIW	IHEXIU	IHEXIS	IHEXIL	IHEXIB	IHEVTB	IHEVSC	IHEVPE	IHEVPC	IHEVPB	IHEVPA	IHEVFE	IHEVFD	IHEVFC	IHEVFB	IHEVFA	IHEVCS	IHEVCA	IHEUPA	IHENTZ	IHETNW	IHETNS	IHETNL	IHETHS	IHETHL	IHESVC	IHESQZ	IHESQW	IHESQS	
CALL																																									CALL
CALLERR																																									CALLERR
CALRTS																																									CALRTS
CKFCB																																									CKFCB
CSVC																																									CSVC
DED																																									DED
DNODE																																									DNODE
EXPG																																									EXPG
FAREA																																									FAREA
FIB																																									FIB
GCURR																																									GCURR
GENER																																									GENER
GETKN																																									GETKN
GFRST																																									GFRST
GNEXT																																									GNEXT
GNODE																																									GNODE
GPREV																																									GPREV
GTRD																																									GTRD
IHEADC																																									IHEADC
IHEBRA		●				●			●	●																		●				●			●	●		●	●		IHEBRA
IHEBXT							●	●	●	●																						●			●	●		●	●		IHEBXT
IHECAL																																									IHECAL
IHEDCV	●	●	●	●	●	●	●	●	●	●								●										●	●	●	●	●	●	●	●	●		●	●	●	IHEDCV
IHEDIF																				●	●																				IHEDIF
IHEERRCD							●	●	●	●	●	●		●	●				●	●											●		●	●						●	IHEERRCD
IHEEXT		●				●					●	●	●						●	●		●		●		●	●	●		●								●	●		IHEEXT
IHEFCB																																									IHEFCB
IHEFCIB																																									IHEFCIB
IHEFROM																																									IHEFROM
IHEHDR	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	IHEHDR
IHEIOD																																									IHEIOD
IHELBE																																									IHELBE
IHELIB	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	IHELIB
IHEMOPP																																									IHEMOPP
IHENAME																																									IHENAME
IHEOPENT																																									IHEOPENT
IHEPCH																																									IHEPCH
IHEPRV																																									IHEPRV
IHERET																																									IHERET
IHERST	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●		●		●	●				●	●		●	●	●	●	●	●	●	●	●		●	●	●	IHERST
IHESAV	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●		●										●	●	●	●	●	●	●	●	●		●	●	●	IHESAV
IHESCV											●	●	●																												IHESCV
IHESDR																																									IHESDR
IHESYM																																									IHESYM
IHETLR	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	IHETLR
IHEZAP																																					●				IHEZAP
INODE																																									INODE
INST																																									INST
LIBDEF																																									LIBDEF
MNODE																																									MNODE
RCON																																									RCON
READDISK																																									READDISK
READTERM																																									READTERM
RFIB																																									RFIB
RTSSVC	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●		●	●	●	RTSSVC
SETDISK																																									SETDISK
SETERRCD																																									SETERRCD
SETFLCA																																									SETFLCA
SETSDV																																									SETSDV
SKPTK																																									SKPTK
SYMDEF																																									SYMDEF
TALLY																																									TALLY
TGENER																																									TGENER
	IHEXGZ	IHEYGX	IHEYGW	IHEYGS	IHEYGL	IHEYGF	IHEXXZ	IHEXXW	IHEXXS	IHEXXL	IHEXIZ	IHEXIW	IHEXIU	IHEXIS	IHEXIL	IHEXIB	IHEVTB	IHEVSC	PHEVPE	IHEVPC	IHEVPB	IHEVPA	IHEVFE	IHEVFD	IHEVFC	IHEVFB	IHEVFA	IHEVCS	IHEVCA	IHEUPA	IHENTZ	IHETNW	IHETNS	IHETNL	IHETHS	IHETHL	IHESVC	IHESQZ	IHESQW	IHESQS	

Called Macros

Figure J-3. Runtime Module-Macro Cross Reference (Page 3 of 3)

MODULE-MODULE CROSS REFERENCE

CALL/360-OS PL/I compiler modules are discussed in detail in Section 3 of this manual. For each module, other routines called by the module are listed under "Routines Called". This interrelationship of modules is summarized in Figure J-4.

In Figure J-4, names are given exactly as they appear in program coding. That is, names which do not follow the prescribed naming conventions (begin with $) are identified.

Modules Called \ Calling Modules	ABAL	ACGEN	ANCRE	APRC	AREXP	ASIDX	ATKN	BEGIN	BGET	BLPRC	BONSA	BPUT	BRNH	CALL	CATEG	$CCONT	CERR	CIF	CNT	CON	CRVT	CSTOP	DCLGN	DDS	$DEXP	DIOS	DOCS	DOG	
ABAL			●	●																			●						ABAL
ACGEN																			●										ACGEN
ANCRE				●																			●						ANCRE
APRC																			●										APRC
AREXP			●									●	●					●										●	AREXP
ASIDX							●																						ASIDX
ATKN																			●										ATKN
BEGIN																			●										BEGIN
BGET																			●										BGET
BLPRC				●				●											●										BLPRC
BONSA																				●	●								BONSA
BPUT																			●										BPUT
BRNH																			●										BRNH
CALL																			●										CALL
CATEG																			●	●									CATEG
$CCONT																													$CCONT
CERR		●	●						●						●														CERR
CIF																			●										CIF
CNT																●													CNT
CON																			●										CON
CRVT																			●										CRVT
CSTOP																			●										CSTOP
DCLGN																			●										DCLGN
DDS									●			●																	DDS
$DEXP																												●	$DEXP
DIOS									●			●																	DIOS
DOCS																													DOCS
DOG																			●					●					DOG
DRET																			●										DRET
EDGN																			●					●					EDGN
ENDES																			●										ENDES
ENDON																			●										ENDON
EXPND		●																						●					EXPND
EYPND		●																						●					EYPND
FIND				●			●		●			●				●			●										FIND
FLG																													FLG
FMT																			●										FMT
FNB							●																						FNB
FORI																													FORI
FPDL									●			●																	FPDL
FSYM										●			●										●						FSYM
FVAR									●		●	●		●										●					FVAR
	ABAL	ACGEN	ANCRE	APRC	AREXP	ASIDX	ATKN	BEGIN	BGET	BLPRC	BONSA	BPUT	BRNH	CALL	CATEG	$CCONT	CERR	CIF	CNT	CON	CRVT	CSTOP	DCLGN	DDS	$DEXP	DIOS	DOCS	DOG	

Figure J-4. Compilation Module-Module Cross Reference (Page 1 of 6)

Calling Modules

Modules Called	ABAL	ACGEN	ANCRE	APRC	AREXP	ASIDX	ATKN	BEGIN	BGET	BLPRC	BONSA	BPUT	BRNH	CALL	CATEG	$CCONT	CERR	CIF	CNT	CON	CRVT	CSTOP	DCLGN	DDS	$DEXP	DIOS	DOCS	DOG	
GPUT																	●												GPUT
$GTRIA			●	●				●	●	●		●	●	●				●	●	●	●	●		●	●			●	$GTRIA
$HAINI																													$HAINI
$HCTP																													$HCTP
$HDVTP																													$HDVTP
$HLNTP																													$HLNTP
$HRTLL																													$HRTLL
$HSCAL																													$HSCAL
$HTCR																													$HTCR
$MCWU																													$MCWU
$NATTP																													$NATTP
$NCALL																													$NCALL
$NCONS			●									●								●	●			●	●				$NCONS
$NCSDV																													$NCSDV
$NCVT																													$NCVT
$NEXP		●	●									●	●	●				●						●				●	$NEXP
$NLSIB																●													$NLSIB
$NMULT																													$NMULT
$NOPCV																													$NOPCV
$NOPRT																									●				$NOPRT
$NPRE																													$NPRE
OPEN																			●										OPEN
$OPMZO																													$OPMZO
$SCDV																								●					$SCDV
SVC							●										●												SVC
$TCODE																			●										$TCODE
$TOPR																													$TOPR
$TRIAD			●																										$TRIAD
$VASGA																													$VASGA
$VASGC																													$VASGC
$VCLR																●													$VCLR
$VDSAC																													$VDSAC
$VGTMP																													$VGTMP
$VINSA																													$VINSA
$WBACK																													$WBACK
$WCONT																													$WCONT
$WCTCT			●				●									●		●						●	●			●	$WCTCT
WEXP			●	●		●	●	●		●						●		●	●	●					●			●	WEXP
$WSTEP																													$WSTEP
$XERR	●	●	●	●	●	●	●	●	●	●	●	●	●	●				●	●	●	●	●	●	●		●	●	●	$XERR
	ABAL	ACGEN	ANCRE	APRC	AREXP	ASIDX	ATKN	BEGIN	BGET	BLPRC	BONSA	BPUT	BRNH	CALL	CATEG	$CCONT	CERR	CIF	CNT	CON	CRVT	CSTOP	DCLGN	DDS	$DEXP	DIOS	DOCS	DOG	

Figure J-4. Compilation Module-Module Cross Reference (Page 2 of 6)

Calling Modules

Modules Called	DRET	EDGN	ENDES	ENDON	EXPND	EYPND	FIND	FLG	FMT	FNB	FORI	FPDL	FSYM	FVAR	GPUT	$GTRIA	$HAINI	$HCTP	$HDVTP	$HLNTP	$HRTLL	$HSCAL	$HTCR	$MCWU	$NATTP	$NCALL	$NCONS	$NCSDV
ABAL																												
ACGEN																												
ANCRE																												
APRC																												
AREXP	●							●			●																	
ASIDX										●																		
ATKN																												
BEGIN																												
BGET																												
BLPRC		●							●																			
BONSA																												
BPUT																												
BRNH																												
CALL																												
CATEG																												
$CCONT																												
CERR		●		●									●															
CIF																												
CNT																												
CON																												
CRVT																												
CSTOP																												
DCLGN																												
DDS																												
$DEXP					●			●																				
DIOS																												
DOCS																												
DOG																												
DRET																												
EDGN						●		●																				
ENDES		●																										
ENDON		●																										
EXPND																												
EYPND																												
FIND																												
FLG									●			●																
FMT																												
FNB		●																										
FORI								●																				
FPDL																												
FSYM																												
FVAR								●																				

Figure J-4. Compilation Module-Module Cross Reference (Page 3 of 6)

Calling Modules

Modules Called	DRET	EDGN	ENDES	ENDON	EXPND	EYPND	FIND	FLG	FMT	FNB	FORI	FPDL	FSYM	FVAR	GPUT	$GTRIA	$HAINI	$HCTP	$HDVTP	$HLNTP	$HRTLL	$HSCAL	$HTCR	$MCWU	$NATTP	$NCALL	$NCONS	$NCSDV	
GPUT																	●				●			●					GPUT
$GTRIA	●	●	●	●	●			●	●		●	●												●				●	$GTRIA
$HAINI																								●					$HAINI
$HCTP																								●					$HCTP
$HDVTP																								●					$HDVTP
$HLNTP																								●					$HLNTP
$HRTLL																								●					$HRTLL
$HSCAL																								●					$HSCAL
$HTCR																								●					$HTCR
$MCWU																													$MCWU
$NATTP																													$NATTP
$NCALL																													$NCALL
$NCONS								●			●																		$NCONS
$NCSDV																													$NCSDV
$NCVT																											●		$NCVT
$NEXP	●							●			●																		$NEXP
$NLSIB																													$NLSIB
$NMULT																													$NMULT
$NOPCV																													$NOPCV
$NOPRT																													$NOPRT
$NPRE																													$NPRE
OPEN																													OPEN
$OPMZO																													$OPMZO
$SCDV																													$SCDV
SVC		●													●		●				●		●	●					SVC
$TCODE		●							●			●																	$TCODE
$TOPR																													$TOPR
$TRIAD																										●		●	$TRIAD
$VASGA																													$VASGA
$VASGC																													$VASGC
$VCLR																													$VCLR
$VDSAC																													$VDSAC
$VGTMP																													$VGTMP
$VINSA																													$VINSA
$WBACK	●	●																											$WBACK
$WCONT																													$WCONT
$WCTCT		●	●	●	●	●						●														●		●	$WCTCT
WEXP		●			●			●			●			●		●											●	●	WEXP
$WSTEP																		●		●		●							$WSTEP
$XERR	●	●						●	●	●	●															●			$XERR
	DRET	EDGN	ENDES	ENDON	EXPND	EYPND	FIND	FLG	FMT	FNB	FORI	FPDL	FSYM	FVAR	GPUT	$GTRIA	$HAINI	$HCTP	$HDVTP	$HLNTP	$HRTLL	$HSCAL	$HTCR	$MCWU	$NATTP	$NCALL	$NCONS	$NCSDV	

Figure J-4. Compilation Module-Module Cross Reference (Page 4 of 6)

Calling Modules

Modules Called	$NCVT	$NEXP	$NLSIB	$NMULT	$NOPCV	$NOPRT	$NPRE	OPEN	$OPMZO	$SCDV	SVC	$TCODE	$TOPR	$TRIAD	$VASGA	$VASGC	$VCLR	$VDSAC	$VGTMP	$VINSA	$WBACK	$WCONT	$WCTCT	WEXP	$WSTEP	$XERR			
ABAL																													ABAL
ACGEN																													ACGEN
ANCRE																													ANCRE
APRC																													APRC
AREXP								●																					AREXP
ASIDX																													ASIDX
ATKN																						●							ATKN
BEGIN																													BEGIN
BGET																													BGET
BLPRC																													BLPRC
BONSA																													BONSA
BPUT																													BPUT
BRNH																													BRNH
CALL																													CALL
CATEG																													CATEG
$CCONT																													$CCONT
CERR																						●							CERR
CIF																													CIF
CNT																													CNT
CON																													CON
CRVT																													CRVT
CSTOP																													CSTOP
DCLGN																													DCLGN
DDS																													DDS
$DEXP																													$DEXP
DIOS																													DIOS
DOCS								●																					DOCS
DOG																													DOG
DRET																													DRET
EDGN																													EDGN
ENDES																													ENDES
ENDON																													ENDON
EXPND																													EXPND
EYPND																													EYPND
FIND																													FIND
FLG																													FLG
FMT																													FMT
FNB																						●							FNB
FORI																													FORI
FPDL																													FPDL
FSYM																													FSYM
FVAR		●						●																					FVAR
	$NCVT	$NEXP	$NLSIB	$NMULT	$NOPCV	$NOPRT	$NPRE	OPEN	$OPMZO	$SCDV	SVC	$TCODE	$TOPR	$TRIAD	$VASGA	$VASGC	$VCLR	$VDSAC	$VGTMP	$VINSA	$WBACK	$WCONT	$WCTCT	WEXP	$WSTEP	$XERR			

Figure J-4. Compilation Module-Module Cross Reference (Page 5 of 6)

Calling Modules

Modules Called	$NCVT	$NEXP	$NLSIB	$NMULT	$NOPCV	$NOPRT	$NPRE	OPEN	$OPMZO	$SCDV	SVC	$TCODE	$TOPR	$TRIAD	$VASGA	$VASGC	$VCLR	$VDSAC	$VGTMP	$VINSA	$WBACK	$WCONT	$WCTCT	WEXP	$WSTEP	$XERR			
GPUT																				●		●				●			GPUT
$GTRIA								●																					$GTRIA
$HAINI																													$HAINI
$HCTP																													$HCTP
$HDVTP																													$HDVTP
$HLNTP																													$HLNTP
$HRTLL																													$HRTLL
$HSCAL																													$HSCAL
$HTCR																													$HTCR
$MCWU																						●							$MCWU
$NATTP						●																							$NATTP
$NCALL					●	●	●																						$NCALL
$NCONS		●			●	●						●	●	●						●									$NCONS
$NCSDV		●																											$NCSDV
$NCVT												●																	$NCVT
$NEXP								●																					$NEXP
$NLSIB						●						●	●							●									$NLSIB
$NMULT						●																							$NMULT
$NOPCV		●				●	●																						$NOPCV
$NOPRT		●																											$NOPRT
$NPRE		●				●																							$NPRE
OPEN																													OPEN
$OPMZO												●																	$OPMZO
$SCDV												●																	$SCDV
SVC												●								●		●		●					SVC
$TCODE								●																					$TCODE
$TOPR												●																	$TOPR
$TRIAD		●		●	●	●																							$TRIAD
$VASGA																		●		●									$VASGA
$VASGC																				●									$VASGC
$VCLR																													$VCLR
$VDSAC																●				●									$VDSAC
$VGTMP												●				●				●									$VGTMP
$VINSA												●																	$VINSA
$WBACK		●				●						●						●	●	●									$WBACK
$WCONT																													$WCONT
$WCTCT		●			●	●	●					●																	$WCTCT
WEXP	●	●			●	●				●		●		●					●										WEXP
$WSTEP		●				●						●										●							$WSTEP
$XERR	●	●			●	●	●	●				●																	$XERR
	$NCVT	$NEXP	$NLSIB	$NMULT	$NOPCV	$NOPRT	$NPRE	OPEN	$OPMZO	$SCDV	SVC	$TCODE	$TOPR	$TRIAD	$VASGA	$VASGC	$VCLR	$VDSAC	$VGTMP	$VINSA	$WBACK	$WCONT	$WCTCT	WEXP	$WSTEP	$XERR			

Figure J-4. Compilation Module-Module Cross Reference (Page 6 of 6)

READER'S COMMENT FORM

CALL/360-OS PL/I

Systems Manual

GY20-0570-1

Please comment on the usefulness and readability of this publication, suggest additions and deletions, and list specific errors and omissions (give page numbers). All comments and suggestions become the property of IBM. If you wish a reply, be sure to include your name and address.

COMMENTS

fold

fold

fold

fold

- Thank you for your cooperation. No postage necessary if mailed in the U.S.A.
  FOLD ON TWO LINES, STAPLE AND MAIL.

YOUR COMMENTS PLEASE...

Your comments on the other side of this form will help us improve future editions of this publication. Each reply will be carefully reviewed by the persons responsible for writing and publishing this material.

Please note that requests for copies of publications and for assistance in utilizing your IBM system should be directed to your IBM representative or the IBM branch office serving your locality.

fold fold

fold fold

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]

GY20-0570-1

CALL/360-OS PL/I Systems Manual Printed in U.S.A. GY20-0570-1

IBM®

International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, New York 10601
(USA only)

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
(International)