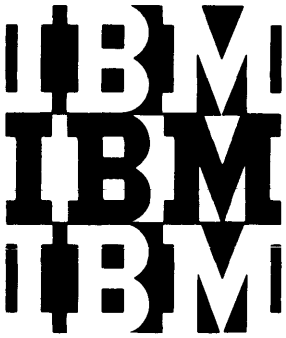


File No. S360-25(OS)
Order No. GY28-6642-5

Program Logic

IBM System/360 Operating System

FORTRAN IV (H) Compiler

Program Logic Manual

Program Number 360S-FO-500

This publication describes the internal design of the IBM System/360 Operating System FORTRAN IV (H) compiler program. This compiler transforms source modules written in the FORTRAN IV language into object modules that are suitable for input to the linkage editor for subsequent execution on System/360. At the user's option, the compiler optimizes its object modules so that they can be executed with improved efficiency.

This manual is directed to IBM customer engineers who are responsible for program maintenance. It can be used to locate specific areas of the program and to relate these areas to the corresponding program listings. Because program logic information is not necessary for program operation and use, distribution of this manual is restricted to persons with program-maintenance responsibilities.

Sixth Edition (October 1972)

This is a reprint of GY28-6642-4 incorporating changes issued in Technical Newsletter GN28-0594, dated July 1, 1971. This edition does not obsolete the previous edition and the associated Technical Newsletter.

The specifications contained in this publication as amended by Technical Newsletter GN28-0594, dated July 1, 1971, correspond to Release 20.1 of the IBM System/360 Operating System.

Changes are periodically made to the specifications herein; any such changes will be reported in subsequent revisions or Technical Newsletters.

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

Address comments concerning the contents of this publication to IBM Corporation, Programming Publications, 1271 Avenue of the Americas, New York, New York 10020.

This publication provides information describing the internal organization and operation of the FORTRAN IV (H) compiler. It is part of an integrated library of IBM System/360 Operating System Program Logic Manuals. Other publications required for an understanding of the FORTRAN IV (H) compiler are:

IBM System/360: Principles of Operation, Order No. GA22-6821

IBM System/360 Operating System:

FORTRAN IV Language, Order No. GC28-6515

Introduction to Control Program Logic, Program Logic Manual, Order No. GY28-6605

FORTRAN IV (G and H) Programmer's Guide, Order No. GC28-6817

Although not required, the following publications are related to this publication and should be consulted:

IBM System/360 Operating System:

Sequential Access Methods, Program Logic Manual, Order No. GY28-6604

Concepts and Facilities, Order No. GC28-6535

Supervisor and Data Management Macro Instructions, Order No. GC28-6647

Linkage Editor and Loader, Order No. GC28-6538

Linkage Editor, Program Logic Manual, Order No. GY28-6610

System Generation, Order No. GC28-6554

This manual consists of two sections.

Section 1 is an introduction that describes the FORTRAN IV (H) compiler as a whole, including its relationship to the operating system. The major components of the compiler and the relationships among them are also described.

Section 2 consists of a discussion of the major components. Each component is discussed in terms of its functions; the level of detail provided is sufficient to enable the reader to understand the general operation of the component. In the discussion of each function of a component, the routines that implement that function are identified by name. The inclusion of a compound form of the routine names provides a frame of reference for the comments and coding supplied in the program listing. The program listing for each identified routine appears on the microfiche card having the second portion of the compound name of that routine in its heading. For example, the routine referred to in this manual as STALL-IEKGST is listed on the microfiche card headed IEKGST. This section also discusses common data, such as tables, blocks, and work areas, but only to the extent required to understand the logic of the components. Flowcharts and routine directories are included at the end of this section.

Following Section 2 are a number of appendixes, which contain descriptions of tables used by the compiler, intermediate text formats, the overlay structure of the compiler, object-time library subprograms, and other reference material.

If more detailed information is required, the reader should refer to the comments and coding in the FORTRAN IV (H) program listing.

ILLUSTRATIONS

FIGURES

TABLES

CHARTS

This section contains general information describing the purpose of the FORTRAN IV (H) compiler, its relationship to the operating system, its input/output data flow, its organization, and its overlay structure.

PURPOSE OF THE COMPILER

The IBM System/360 Operating System FORTRAN IV (H) compiler transforms source modules written in the FORTRAN IV language into object modules that are suitable for input to the linkage editor for subsequent execution on the System/360. At the user's option, the compiler produces optimized object modules (modules that can be executed with improved efficiency).

THE COMPILER AND OPERATING SYSTEM/360

The FORTRAN IV (H) compiler is a processing program that communicates with the System/360 Operating System control program for input/output and other services. A general description of the control program is given in the publication IBM System/360 Operating System: Introduction to Control Program Logic, Program Logic Manual, Form Y28-6605.

A compilation, or a batch of compilations, is requested using the job statement (JOB), the execute statement (EXEC), and data definition statements (DD). Cataloged procedures may also be used. A discussion of FORTRAN IV compilation and the available cataloged procedures is given in the publication IBM System/360 Operating System: FORTRAN IV (G and H) Programmer's Guide, Form C28-6817.

The compiler receives control from the calling program (e.g., job scheduler or another program that calls, links to, or connects the compiler). Once the compiler receives control, it communicates with the control program through the FORTRAN system director, a part of the compiler that controls compiler processing. After compiler processing is completed, control is returned to the calling program.

INPUT/OUTPUT DATA FLOW

The source modules to be compiled are read in from the SYSIN data set. Compiler output is placed on the SYSLIN, SYSPRINT, SYSPUNCH, SYSUT1, or SYSUT2 data set, depending on the options specified by the FORTRAN programmer. (The SYSPRINT data set is always required for compilation.)

The overall data flow and the data sets used for the compilation are illustrated in Figure 1.

COMPILER ORGANIZATION

The IBM System/360 Operating System FORTRAN IV (H) compiler consists of the FORTRAN system director, four logical processing phases (phases 10, 15, 20, and 25), and an error-handling phase (phase 30).

Control is passed among the phases of the compiler via the FORTRAN system director. After each phase has been executed, the FORTRAN system director determines the next phase to be executed, and calls that phase. The flow of control within the compiler is illustrated in Chart 00. (Charts are located at the end of Section 2.)

The components of the compiler operating together produce an object module from a FORTRAN source module. The object module is acceptable as input to the linkage editor, which prepares object modules for relocatable loading and execution.

The object module consists of control dictionaries (external symbol dictionary and relocation dictionary), text (representing the actual machine instructions and data), and an END statement. The external symbol dictionary (ESD) contains the external symbols that have been defined or referred to in the source module. The relocation dictionary (RLD) contains information about address constants in the object module.

The functions of the components of the compiler are described in the following paragraphs.

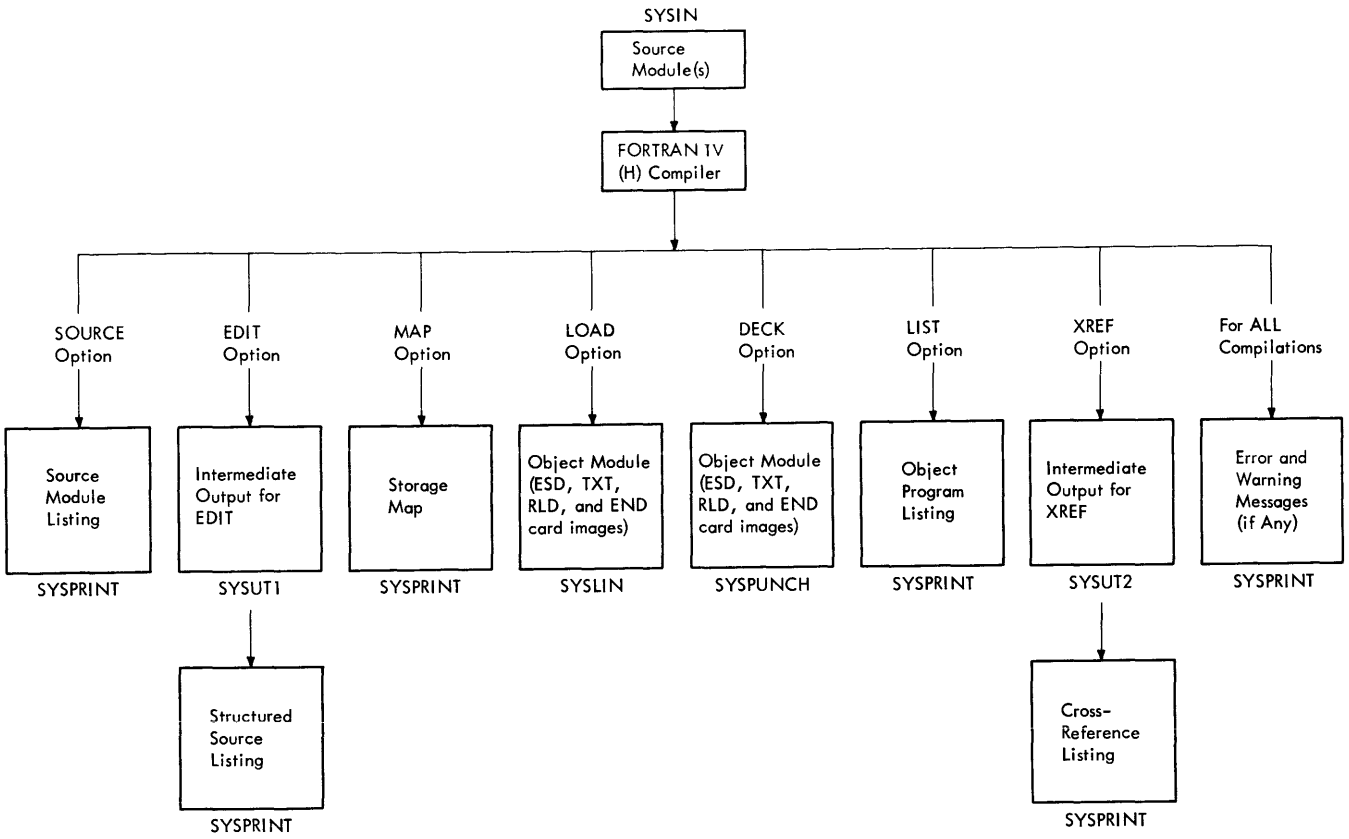


Figure 1. Input/Output Data Flow

FORTRAN SYSTEM DIRECTOR

The FORTRAN system director (FSD) controls compiler processing. It initializes compiler operation, calls the phases for execution, and distributes and keeps track of the main storage used during the compilation. In addition, the FSD receives the various input/output requests of the compiler phases and submits them to the control program.

PHASE 10

Phase 10 accepts as input (from the SYSIN data set) the individual source statements of the source module. If a source module listing is requested, the source statements are recorded on the SYSPRINT data set. If the XREF option is selected, a two-part cross reference is recorded on the SYSPRINT data set immediately following the source listing. If the EDIT option is selected, the source statements are recorded on the SYSUT1 data set, which phase 20 uses as input to produce a structured source listing. If the ID option is selected, calls and function references are assigned an internal statement number (ISN).

Phase 10 converts each source statement into a form usable as input by succeeding phases. This usable input consists of an intermediate text representation (in operator-operand pair format) of each source statement. In addition, phase 10 makes entries in an information table for the variables, constants, literals, statement numbers, etc., that appear in the source statements. Phase 10 also places data about COMMON and EQUIVALENCE statements in the information table so that main storage space can be allocated correctly in the object module. During this conversion process, phase 10 also analyzes the source statements for syntactical errors. If errors are encountered, phase 10 passes to phase 30 (by making entries in an error table) the information needed to print the appropriate error messages.

PHASE 15

Phase 15 gathers additional information about the source module and modifies some

intermediate text entries to facilitate optimization by phase 20 and instruction generation by phase 25. Phase 15 is divided into two segments that perform the following functions:

- The first segment translates phase 10 intermediate text entries (in operator-operand pair format) representing arithmetic operations into a four-part format, which is needed for optimization by phase 20 and instruction-generation by phase 25. This part of phase 15 also gathers information about the source module that is needed for optimization by phase 20.
- The second segment of phase 15 assigns relative addresses and, where necessary, address constants to the named variables and constants in the source module. This segment also converts phase 10 intermediate text (in operator-operand pair format) representing DATA statements to a variable-initial value format, which makes later assignment of a constant value to a variable easier.

Phase 15 also passes to phase 30 the information needed to print appropriate messages for any errors detected during phase 15 processing. (This is done by making entries in the error table.)

PHASE 20

Phase 20 processing depends on whether or not optimization has been requested and, if so, the optimization level desired.

If no optimization is specified, phase 20 assigns registers for use during execution of the object module. However, phase 20 does not take full advantage of all registers and makes no effort to keep frequently used quantities in registers to eliminate the need for some machine instructions.

If the first level of optimization is specified, phase 20 uses all available registers and keeps frequently used quantities in registers wherever possible. Phase 20 takes other measures to reduce the size of the object module, and provides information about operands to phase 25.

If the second level of optimization is specified, phase 20 uses other techniques to make a more efficient object module. The net result of these procedures is to eliminate unnecessary instructions and to eliminate needless execution of instructions.

If both the EDIT option and the second level of optimization are selected, phase 20 produces a structured source program listing on the SYSPRINT data set.

PHASE 25

Phase 25 produces an object module from the combined output of the preceding phases of the compiler.

The text information (instructions and data resulting from the compilation) is in a relocatable machine language format. It may contain unresolved external symbolic cross references (i.e., references to symbols that do not appear in the source module). The external symbol dictionary contains the information required by the linkage editor to resolve external symbolic cross references, and the relocation dictionary contains the information needed by the linkage editor to relocate the absolute text information.

Phase 25 places the object module resulting from the compilation on the SYSLIN data set if the LOAD option is specified, and on the SYSPUNCH data set if the DECK option is specified. Phase 25 produces an object module listing on the SYSPRINT data set if the LIST option is specified. In addition, phase 25 produces a storage map if the MAP option is specified.

PHASE 30

Phase 30 is called after phase 25 processing is completed only if errors are detected by previous phases. Phase 30 records messages describing the detected errors on the SYSPRINT data set.

STRUCTURE OF THE COMPILER

The FORTRAN IV (H) compiler is structured in a planned overlay fashion, which consists of 13 segments. One of these segments constitutes the FORTRAN system director and is the root segment of the planned overlay structure. Each of the remaining 12 segments constitutes a phase or a logical portion of a phase. A detailed discussion of the compiler's planned overlay structure is given in Appendix F.

SECTION 2: DISCUSSION OF MAJOR COMPONENTS

The following paragraphs and associated flowcharts at the end of this section describe the major components of the FORTRAN IV (H) compiler. Each component is described to the extent necessary to explain its function(s) and its general operation.

FORTRAN SYSTEM DIRECTOR

The FORTRAN system director (FSD) controls compiler processing; its overall logic is illustrated in Chart 01. (For a complete list of FSD subroutines, see Table 6.) The FSD receives control from the job scheduler if the compilation is defined as a job step in an EXEC statement. The FSD may also receive control from another program through use of one of the system macro instructions (CALL, LINK, or ATTACH).

The FSD:

- Initializes the compiler.
- Loads the compiler phases.
- Distributes storage to the phases.
- Processes input/output requests.
- Generates entry code (initialization instructions) for main programs, subprograms, and subprogram secondary entries.
- Deletes compilation.
- Terminates compilation.

COMPILER INITIALIZATION

The initialization of compiler processing by the FSD consists of three steps:

- Parameter processing.
- Storage acquisition.
- Data field initialization.

Parameter Processing

When the FSD is given control, the address of a parameter list is contained in a general register. If the compiler receives control as a result of either an EXEC statement in a job step or an ATTACH or CALL macro instruction in another program, the parameter list has a single entry, which is a pointer to the main storage area containing an image of the options (e.g., SOURCE, MAP) specified for the compilation. If the compiler receives control as a result of a LINK macro instruction in another program, the parameter list may have a second entry, which is a pointer to the main storage area containing substitute ddnames (i.e., ddnames that the user wishes to substitute for the standard ones of SYSIN, SYSPRINT, SYSPUNCH, SYSLIN, SYSUT1, and SYSUT2.

COMPILER OPTIONS: To determine the options specified for the compilation and to inform the various compiler phases of these options, the FSD scans and analyzes the storage area containing their images and sets indicators to reflect the ones specified. These indicators are placed into the communication table -- IEKAAA (see Appendix A, "Communication Table") during data field initialization. The various compiler phases have access to the communication table and, from the indicators contained in it, can determine which options have been selected for the compilation.

SUBSTITUTE DDNAMES: If the user wishes to substitute ddnames for the standard ones, the FSD must establish a correspondence between the DD statements having the substitute ddnames and the DCBs (Data Control Blocks) associated with the ddnames to be replaced. To establish this necessary correspondence, the FSD scans the storage area containing the substitute ddnames, and enters each such ddname into the DCBDDNM field of the DCB associated with the standard ddname it is to replace.

Storage Acquisition

The FSD issues GETMAIN's to obtain main storage for work and table areas the compiler will need. Usually, the FSD acquires the entire remaining region (MVT), partition (MFT), or machine (PCP). However, if the user has included a SIZE parameter on his EXEC card, the FSD acquires main storage equal (approximately) to this figure minus compiler code size.

Data Field Initialization

Data field initialization affects the communication table, which is a central gathering area used to communicate information among the phases of the compiler. The table contains information such as:

- User specified options.
- Pointers indicating the next available locations within the various storage areas.
- Pointers to the initial entries in the various types of chains (see "Appendix A, Information Table" and "Appendix B, Intermediate Text").
- Name of the source module being compiled.
- An indication of the phase currently in control.

The various fields of the communication table, which are filled during a compilation, must be initialized before the next compilation. To initialize this region, the FSD clears it and places the option indicators into the fields reserved for them.

PHASE LOADING

The FSD loads and passes control to each phase of the compiler by means of a standard calling sequence. The execution of the call causes control to be passed to the overlay supervisor, which calls program fetch to read in the phase. Control is then returned to the overlay supervisor, which branches to the phase. The phases are called for execution in the following sequence: phase 10, phase 15, phase 20, and phase 25. However, if errors are detected by previous phases, phase 30 is called after the completion of phase 25 processing.

STORAGE DISTRIBUTION (CHART 02)

Phases 10, 15, and 20 require main storage space in which to construct the information table (see Appendix A, "Information Table") and to collect intermediate text entries. These phases obtain this storage space by submitting requests to the FSD (at entry point IEKAGC), which allocates the required space, if available, and returns to the requesting phase pointers to both the beginning and end of the allocated storage space.

Phase 10 Storage

Phase 10 can use all of the available storage space for building the information table and for collecting text entries. At each phase 10 request for main storage in which to collect text entries or build the information table, the FSD reallocates a portion (i.e., a subblock) of the storage for text collection, and returns to phase 10 either via the communication table or the storage area P10A-IEKCAA (depending upon the type of text to be collected in the subblock; see Appendix B, "Phase 10 Intermediate Text") pointers to both the beginning and end of the allocated storage space. If the subblock is allocated for phase 10 normal text or for the information table, the pointers are returned in the communication table. If the subblock is allocated for a phase 10 text type other than normal text, the pointers are returned via the storage area P10A-IEKCAA. After the storage has been allocated, the FSD adjusts the end of the information table downward by the size of the allocated subblock. This process is repeated for each phase 10 request for main storage space.

Subblocks to contain phase 10 text or dictionary entries are allocated in the order in which requests for main storage are received. (When phase 10 completely fills one subblock with text entries, it requests another.) A request for a subblock to contain a particular type of entry may immediately follow a request for a subblock to contain another type of entry. Consequently, subblocks allocated to contain the same type of entries may be scattered throughout main storage. The FSD must keep track of the subblocks so that, at the completion of phase 10 processing, unused or unnecessary storage may be allocated to phase 15.

Phase 15 Storage

Phase 15, in collecting the text or dictionary entries that it creates, can use only those portions of main storage that are (1) unused by phase 10, or (2) occupied by phase 10 normal text entries that have been processed by phase 15. The FSD first allocates all unused storage (if necessary) to phase 15. If this is not sufficient,

the FSD then allocates the storage occupied by phase 10 normal text entries that have undergone phase 15 processing. If either of these methods of storage allocation fails to provide enough storage for phase 15, the compilation is terminated.

Pointers to both the beginning and end of the allocated subblock portion are passed to phase 15 via the communication table. If an additional request is received after the last subblock portion is allocated, the FSD determines the last phase 10 normal text entry that was processed by phase 15. The FSD then frees and allocates to phase 15 the portion of storage occupied by phase 10 normal text entries between the first such text entry and the last entry processed by phase 15.

Phase 15 Storage Inventory: After the processing of PHAZ15, the first segment of phase 15, is completed, the FSD recovers the subblocks that were allocated to phase 10 normal text. These subblocks are chained as extensions to the storage space available at the completion of PHAZ15 processing. The chain, which begins in the FSD pointer table, connecting the various available portions of storage is scanned and when a zero pointer field is encountered, a pointer to the first subblock allocated to phase 10 normal text is placed into that field. The chain connecting the various subblocks allocated to phase 10 normal text is then scanned and when a zero pointer field is encountered, a pointer to the first subblock allocated to SF skeleton text is placed into that field. Once the subblocks are chained in this manner, they are available for allocation to CORAL, the second segment of phase 15, and to phase 20.

After the processing of CORAL is completed, the FSD likewise recovers the subblocks allocated for phase 10 special text. The chain connecting the various portions of available storage space is scanned and when a zero pointer field is encountered, a pointer to the first subblock allocated for phase 10 special text is placed into that field. After the subblocks allocated for phase 10 special text are linked into the chain as described above, they, as well as all other portions of storage space in the chain, are available for allocation to phase 20.

Phase 20 Storage

Each phase 20 request for storage space is satisfied with a portion of storage available at the completion of CORAL processing. The portions of storage are allocated to phase 20 in the order in which they are chained. Pointers to both the beginning and end of the storage allocated to phase 20 for each request are placed into the communication table.

INPUT/OUTPUT REQUEST PROCESSING

The FSD routine IEKFCOMH receives the input/output requests of the compiler phases and submits them to QSAM (Queued Sequential Access Method) for implementation (see the publication IBM System/360 Operating System: Sequential Access Methods, Program Logic Manual, Form Y28-6604).

Request Format

Phase requests for input/output services are made in the form of READ/WRITE statements requiring a FORMAT statement. The format codes that can appear in the FORMAT statement associated with such READ/WRITE requests are a subset of those available in the FORTRAN IV language. The subset consists of the following codes: Iw (output only), Tw, Aw, wX, wH, and Zw (output only).

Request Processing

To process input/output requests from the compiler phases, the FSD performs a series of operations, which are a subset of those carried out by the IEKFCOMH/IEKFIOCS Library routines to implement sequential READ/WRITE statements requiring a format.

GENERATION OF INITIALIZATION INSTRUCTIONS

The FSD subroutine IEKTLOAD works with STALL to generate the machine instructions for entry into a program. These instructions are referred to as initialization instructions and are divided into three catagories:

- Entry coding for a main program.
- Entry coding for subprograms with no secondary entry points.
- Main entry coding for subprograms with secondary entry points.

Once generated, these instructions are entered into TXT records (see "Phase 25, Text Information" for a discussion of TXT records).

Entry Coding for a Main Program

The initialization instructions generated by subroutine IEKTLOAD for a main program perform the following functions:

- Branch past the eight-byte name field to the store multiple instruction.
- Save the contents of registers 14 through 12 in the save area of the calling program.
- Load the address of the prologue into register 2 and the address of the save area into register 3.
- Store the location of the called program's save area into the third word of the calling program's save area.
- Store the location of the calling program's save area into the second word of the called program's save area.
- Branch to the prologue. (For an explanation of prologue and epilogue, see "Phase 25, Prologue and Epilogue Generation.")

The prologue instructions perform the following functions:

- Load register 12, if register 12 is used.
- Load register 15 for the following call to IBCOM.
- Call IBCOM for main program initialization.
- Load register 13 with the address of the called program's save area.
- Branch to the first instruction in the body of the program.

Entry Coding for Subprograms with No Secondary Entry Points

The initialization instructions generated by subroutine IEKTLOAD for the entry points into a subprogram with no secondary entry points perform the following functions:

- Branch past the eight-byte name field to the store multiple instruction.
- Save the contents of general registers 14 through 12 in the save area of the calling program.
- Load the address of the calling program's save area into register 4.
- Load the address of the prologue into register 12 and the address of the save area into register 13.
- Store the location of the calling program's save area into the second word of the called program's save area.
- Store the location of the called program's save area into the third word of the calling program's save area.
- Branch to the prologue. (For an explanation of prologue and epilogue, see "Phase 25, Prologue and Epilogue Generation.")

The prologue instructions perform the following functions:

- Initialize call by value arguments (if any) and also initialize adcons for call by name arguments (if any).
- Branch to the first instruction in the body of the called program.

Main Entry Coding for Subprograms with Secondary Entry Points

The initialization instructions generated by subroutine IEKTLOAD for the main entry point into a subprogram with secondary entry points perform the following functions:

- Branch past the eight-byte name field to the store multiple instruction.
- Save the contents of registers 14 through 12 in the save area of the calling program.
- Load the address of the prologue into register 2 and the address of the epilogue into register 3.

- Load the location of the calling program's save area into register 4.
- Load the location of the called program's save area into register 13.
- Store the address of the epilogue into the first word of the called program's save area and the location of the calling program's save area into the second word of the called program's save area.
- Store the location of the called program's save area into the third word of the calling program's save area.
- Branch to the prologue.

The main entry prologue instructions (generated by phase 25) perform the same functions described previously under "Entry Coding for Subprograms with No Secondary Entry Points."

Subprogram Secondary Entry Coding

This coding is generated entirely by phase 25 but is mentioned here for completeness. The requirements of secondary entry coding are essentially the same as main entry coding. For this reason many of the main entry instructions are used by phase 25 through an unconditional branch into that section of code. Main entry instructions that precede and include the instruction which loads the prologue and epilogue addresses cannot be used, since each secondary entry point has its own associated prologue and epilogue. Therefore, secondary entry instructions perform the following functions:

- Branch past the eight-byte name field to the store multiple instruction.
- Save the contents of registers 14 through 12 in the save area of the calling program.
- Load the address of the prologue into register 2 and the address of the epilogue into register 3.
- Load register 15 with the address of the instruction in the main entry coding that loads register 4.
- Branch into the main entry coding.

The secondary entry prologue instructions (generated by phase 25) perform the same functions described previously for subprogram main entry coding, except that the branch is directed to the desired entry point in the body of the called program rather than the first instruction.

Subprogram secondary entry coding does not occupy storage within the "Initialization Instructions" section of text information. That section is reserved for:

- Main program entry coding, if the source module being compiled is a main program.
- Subprogram main entry coding, if a subprogram is being compiled.

The secondary entry coding is generated for each occurrence of an ENTRY statement, followed immediately by its associated prologue and epilogue. Secondary entry coding follows the main prologue and epilogue which, in turn, follow the main body of the program. For each additional secondary entry point, equivalent instructions will be generated.

DELETION OF A COMPILATION

The FSD deletes a compilation if an error of error level code 16 (see the publication IBM System/360 Operating System: FORTRAN IV (G and H) Programmer's Guide, Form C28-6817) is detected during the execution of a processing phase.

The phase detecting the error passes control to the FSD at entry point SYSDIR-IEKAA9. If the error was detected by phase 10, the FSD deletes the compilation by having phase 10 read records (without process- ing them) until the END statement is encountered. If the error was encountered in a phase other than phase 10, the FSD simply deletes the compilation.

COMPILER TERMINATION

The FSD terminates compiler processing when an end-of-file is encountered in the input data stream or when a permanent input/output error is encountered. If, after the deletion of a compilation or after a source module has been completely compiled, the first record read by the FSD from the SYSIN data set contains an end-of-file indicator, control is passed to the FSD (at the entry point ENDFILE), which terminates compiler processing by returning control to the operating system. If a permanent error is encountered during the servicing of an input/output request of a

phase, control is passed to the FSD (at entry point IBCOMRTN), which writes a message stating that both the compilation and job step are deleted. The FSD then returns control to the operating system. In either of the above cases, the FSD passes to the operating system as a condition code the value of the highest error level code encountered during compiler processing. The value of the code is used to determine whether or not the next job step is to be performed.

PHASE 10

The FSD reads the first record of the source module and passes its address to phase 10 via the communication table. Phase 10 converts each FORTRAN source statement into usable input to subsequent phases of the compiler; its overall logic is illustrated in Chart 03. Phase 10 conversion produces an intermediate text representation of the source statement and/or detailed information describing the variables, constants, literals, statement numbers, data set reference numbers, etc., appearing in the source statement. During conversion, the source statement is analyzed for syntactical errors.

The intermediate text is a strictly defined internal representation (i.e., internal to the compiler) of a source statement. It is developed by scanning the source statement from left to right and by constructing operator-operand pairs. In this context, operator refers to such elements as commas, parentheses, and slashes, as well as to arithmetic, relational, and logical operators. Operand refers to such elements as variables, constants, literals, statement numbers, and data set reference numbers. An operator-operand pair is a text entry, and all text entries for the operator-operand pairs of a source statement are the intermediate text representation of that statement.

The following six types of intermediate text are developed by phase 10:

- Normal text is the intermediate text representation of source statements other than DATA, NAMELIST, DEFINE FILE, FORMAT, and statement functions.
- Data text is the intermediate text representation of DATA statements and initialization values in type statements.
- Namelist text is the intermediate text representation of NAMELIST statements.
- Define file text is the intermediate text representation of DEFINE FILE statements.
- Format text is the intermediate text representation of FORMAT statements.
- SF skeleton text is the intermediate text representation of statement functions using sequence numbers as operands of the intermediate text entries. The sequence numbers replace the dummy arguments of the statement functions. This type of text is, in effect, a "skeleton" macro instruction.

The various text types are discussed in detail in Appendix B, "Intermediate Text."

The detailed information describing operands includes such facts as whether a variable is dimensioned (i.e., an array) and whether the elements of an array are real, integer, etc. Such information is entered into the information table.

The information table consists of five components, as follows:

- The dictionary contains information describing the constants and variables of the source module.
- The statement number/array table contains information describing the statement numbers and arrays of the source module.
- The common table contains information describing COMMON and EQUIVALENCE declarations.
- The literal table contains information describing the literals of the source module.
- The branch table contains information describing statement numbers that appear in computed GO TO statements.

A detailed discussion of the information table is given in Appendix A, "Information Table."

The intermediate text and the information table complement each other in the actual code generation by the subsequent phases. The intermediate text indicates what operations are to be carried out on specific operands; the information table provides the detailed information describing the operands that are to be processed.

SOURCE STATEMENT PROCESSING

To process source statements, each record (one card image) of the source module is first read into an input buffer by a preparatory subroutine (GETCD-IEKCGC). If a source module listing is requested, the record is recorded on an output data set (SYSPRINT). If both the EDIT option and the second level of optimization (OPT=2) are selected, the record and some control information used by phase 20 to produce a structured source listing are recorded on the SYSUT1 data set. Records are moved to an intermediate buffer until a complete source statement resides in that buffer. Unnecessary blanks are eliminated from the source statement, and the statement is assigned a classification code. A dispatcher subroutine (DSPTCH-IEKCDP) determines from the code which subroutine is to continue processing the source statement. Control is then passed to that subroutine, which converts the source statement to its intermediate text representation and/or constructs information table entries describing its operands (see Table 7 for a list of the subroutines that process each type of statement). After the entire source statement has been processed, the next statement is read and processed as described above. The recognition of the END statement causes phase 10 to complete its processing and return control to the FSD, which then calls phase 15 for execution.

The functions of phase 10 are performed by six groups of subroutines:

- Dispatcher subroutine
- Preparatory subroutine
- Keyword subroutine(s)
- Arithmetic subroutine(s)
- Utility subroutine(s)
- STALL-IEKGST subroutine

Dispatcher Subroutine

The dispatcher subroutine (DSPTCH-IEKCDP) controls phase 10 processing. Upon receiving control from the FSD, the DSPTCH-IEKCDP subroutine initializes phase 10 processing and then calls the preparatory subroutine (GETCD-IEKCGC) to read and prepare the first source statement. After the statement is prepared, control is returned to DSPTCH-IEKCDP, which determines whether or not a statement number is associated with the source statement being processed. If there is a statement number, the XCLASS-IEKDCL subroutine is called to construct a statement number entry (see Appendix A, "Information Table") and a corresponding text entry. DSPTCH-IEKCDP then determines, from the classification code assigned to the source statement (see "Preparatory Subroutine"), which subroutine (either keyword or arithmetic) is to continue the processing of the statement, and passes control to that subroutine. When the source statement is completely processed, control is returned to the DSPTCH-IEKCDP subroutine, which calls the preparatory subroutine to read and prepare the next source statement.

Preparatory Subroutine

The preparatory subroutine (GETCD-IEKCGC) reads each source statement, records it on the SYSPRINT data set if the SOURCE option is selected, and on the SYSUT1 data set if the EDIT option and the second level of optimization are selected, packs and classifies it, and assigns it an internal statement number (ISN).[1] Packing eliminates unnecessary blanks, which may precede the first character, follow the last character, or be imbedded within the source statement. Classifying assigns a code to each type of source statement. The code indicates to the DSPTCH-IEKCDP subroutine which subroutine is to continue processing the source statement. A description of the classifying process, along with figures illustrating the two tables (the keyword pointer table and the keyword table) used in this process, is given in Appendix A, "Classification Tables." The ISN assigned to the source statement is an internal sequence number used to identify the source statement. The source statement and classification information about the source statement reside in the storage areas, NCDIN and NCARD of the phase 10 common area, as illustrated in Figure 2.

---

[1]Logical IF statements are assigned two internal statement numbers. The IF part is given the first number and the "trailing" statement is given the next.

NCARD

Pointer to first character of packed source statement beyond keyword[1]	(1 word)
Internal statement number (ISN)	(1 word)
Statement number indicator (≠0 if present; 0 if not present)	(1 word)
Classification code	(1 word)

NCDIN

Statement number	(5 bytes)
Packed source statement	(n bytes)
Group mark[2]	(1 byte)
[1]For arithmetic statements and statement functions, this field points to the first character of the packed statement. [2]End of statement marker ('4F' in hexadecimal).	

Figure 2. Format of Prepared Source Statement

Keyword Subroutine(s)

A keyword subroutine exists for each keyword source statement. A keyword source statement is any permissible FORTRAN source statement other than an arithmetic statement or a statement function. The function of each keyword subroutine is to convert its associated keyword source statement (in NCDIN) into input usable by subsequent phases of the compiler. These subroutines make use of the utility subroutines and, at times, the arithmetic subroutines in performing their functions. To simplify the discussion of these subroutines, they are divided into two groups:

1. Those that construct only information table entries.

2. Those that construct information table entries and develop intermediate text representations.

Table Entry Subroutines: Only one keyword subroutine belongs to this group (see Table 8). It is associated with a COMMON, DIMENSION, EQUIVALENCE, or EXTERNAL keyword statement.

This subroutine scans its associated statement (in NCDIN) from left to right and constructs appropriate information table entries for each of the operands of the statement. The types of information table entries that can be constructed by these subroutines are:

- Dictionary entries for variables and external names.
- Common block name entries for common block names.
- Equivalence group entries for equivalence groups.
- Equivalence variable entries for the variables in an equivalence group.
- Dimension entries for arrays.

The formats of these entries are given in Appendix A, "Information Table."

Table Entry and Text Subroutines: The keyword subroutines, other than the table entry subroutine, belong to this group (see Table 8). Each of these subroutines converts its associated statement by developing an intermediate text representation of the statement, which consists of text entries in operator-operand pair format, and constructing information table entries for the operands of the statement. The processing performed by these subroutines is similar and is described in the following paragraphs.

Upon receiving control from the DSPTCH-IEKCDP subroutine, the keyword subroutine associated with the keyword statement being processed places a special operator into the text area. This operator is referred to as a primary adjective code and defines the type (e.g., DO, ASSIGN) of the statement. A left-to-right scan of the source statement is then initiated. The first operand is obtained, an information table entry is constructed for the operand and entered into the information table (only if that operand was not previously entered), and a pointer to the entry's location in that table is placed into the text area. The mode (e.g., integer, real) and type (e.g., negative constant, array) of the operand are then placed into text.

Scanning is resumed and the next operator is obtained and placed into the text area. The next operand is then obtained, an information table entry is constructed for the operand and entered into the information table (again, only if that operand was not previously entered), and a pointer to the entry's location is placed into the text entry work area. The mode and type of the operand are placed into the work area. The text entry is then placed into the next available location in

the subblock allocated for text entries of the type being created.

This process is terminated upon recognition of the end of the statement, which is marked by a special text entry. The special text entry contains an end mark operator and the ISN of the source statement as an operand.

Note: Certain keyword subroutines in this group, namely those that process statements that can contain an arithmetic expression (e.g., IF and CALL statements) and those that process statements that contain I/O list items (e.g., READ/WRITE statements), pass control to the arithmetic subroutines to complete the processing of their associated keyword statements.

Arithmetic Subroutine(s)

The arithmetic subroutine or subroutines (see Table 8) receive control from the DSPTCH-IEKCDP subroutine, or from various keyword subroutines. These subroutines make use of the utility subroutines in performing their functions, which are to:

- Process arithmetic statements.
- Process statement functions.
- Complete the processing of certain keyword statements (READ, WRITE, CALL, and IF).

Arithmetic subroutines are processed according to their functions, as follows:

Arithmetic Statement Processing: In processing an arithmetic statement, the arithmetic subroutines develop an intermediate text representation of the statement, and construct information table entries for its operands. These subroutines accomplish this by following a procedure similar to that described for keyword (table entry and text) subroutines.

If one operator is adjacent to another, the first operator does not have an associated operand. In the example A=B(I)+C, the operator + has variable C as its associated operand, whereas the operator) has no associated operand. If an operator has no associated operand, it is assumed that the operand is a zero (null).

Statement Function Processing: In converting a statement function to usable input to subsequent phases of the compiler, the arithmetic subroutines develop an intermediate text representation of the statement function using sequence numbers as replacements for dummy arguments. These subroutines also construct information table entries for those operands that appear to the right of the equal sign and that do not correspond to dummy arguments. The following paragraphs describe the processing of a statement function by the arithmetic subroutines.

When processing a statement function, the arithmetic subroutines:

- Scan the portion of the statement function to the left of the equal sign, obtain each dummy argument, assign each dummy argument a sequence number (in ascending order), and save the dummy arguments and their associated sequence numbers for subsequent use.
- Scan the portion of the statement function to the right of the equal sign and obtain the first (or next) operand.
- Determine whether or not the operand corresponds to a dummy argument. If it does correspond, its associated sequence number is placed into the text area. If it does not correspond, a dictionary entry for the operand is constructed and entered into the information table, and a pointer to the entry's location is placed into the text area. (An opening parenthesis is used as the operator of the first text entry developed for each statement function and a closing parenthesis is used as the operator of the last text entry developed for each statement function.)
- Resume scanning, obtain the next operator, and place it into the text area.
- Obtain the operand to the right of this operator and process it as described above.

Keyword Statement Completion: In addition to processing arithmetic statements and statement functions, the arithmetic subroutines also complete the processing of keyword statements that may contain arithmetic expressions or that contain I/O list items. The keyword subroutine associated with each such keyword statement performs the initial processing of the statement, but passes control to the arithmetic subroutines at the first possible occurrence of an arithmetic expression or an I/O list item. (For example, the keyword subroutine that processes CALL statements passes control to the arithmetic subroutines after it has processed the first opening parenthesis of the CALL statement, because the argument

that follows this parenthesis may be in the form of an arithmetic expression.) The arithmetic subroutines complete the processing of these keyword statements in the normal manner. That is, they develop text entries for the remaining operator-operand pairs and construct information table entries for the remaining operands.

Utility Subroutine(s)

The utility subroutines (see Table 8) aid the keyword, arithmetic, and DSPTCH-IEKCDP subroutines in performing their functions. The utility subroutines are divided into the following groups:

- Entry placement subroutines.
- Text generation subroutines.
- Collection subroutines.
- Conversion subroutines.

Entry Placement Subroutines: The utility subroutines in this group place the various types of entries constructed by the keyword, arithmetic, and DSPTCH-IEKCDP subroutines into the tables or text areas (i.e., subblocks) reserved for them.

Text Generation Subroutines: The utility subroutines in this group generate text entries (supplementary to those developed by the keyword and arithmetic subroutines) that:

- Control the execution of implied DOs appearing in input/output statements.
- Increment DO indexes and test them against their maximum values.
- Signify the end of a source statement.

Collection Subroutines: These utility subroutines perform such functions as gathering the next group of characters (i.e., a string of characters bounded by delimiters) in the source statement being processed, and aligning variable names on a word boundary for comparison to other variable names.

Conversion Subroutines: These utility subroutines convert integer, real, and complex constants to their binary equivalents.

Subroutine STALL-IEKGST (Chart 04)

The STALL-IEKGST subroutine completes phase 10 processing by:

- Generating entry code for the object module.
- Translating phase 10 format text into object code for the object module and freeing space formerly occupied by the format text.
- Checking to see if any literal data text exists and, if it does, generating object code for the literal data text.
- Processing any equivalence entries that were equivalenced before being dimensioned.
- Setting aside space in the object module for the problem program save area and for computed GO TO statement branch tables created by phase 10.
- Checking the statement number section of the information table for undefined statement numbers.
- Rechaining variables in the dictionary by sorting alphabetically the entries in each chain.
- Assigning coordinates based on the usage count set by phase 10 when the OPT option is greater than zero.
- Processing common entries in the information table by computing the displacement of each variable in the common block from the start of the common block.
- Processing equivalence entries in the information table.

Generating FORMAT Code: If the source module contains READ/WRITE statements requiring FORMAT statements, the associated phase 10 format text must be put into a form recognizable by the IHCFCOMH Library routine. The STALL-IEKGST subroutine calls subroutine FORMAT-IEKTFM which develops the necessary format by obtaining the phase 10 intermediate text representation of each FORMAT statement, and translating each element (e.g., H format code and field count) of the statement according to Table 1. The FORMAT-IEKTFM subroutine enters the translated statement along with its relative address into TXT records. It also inserts the relative address of the translated statement into the address

Table 1. FORMAT Statement Translation

FORMAT Specification	Description	Translated Format (in hexadecimal)		
		1st byte	2nd byte	3rd byte
	beginning of statement	02		
n(	group count	04	n	
n	field count	06	n	
nP	scaling factor	08	n*	
Fw.d	F-conversion	0A	w	d
Ew.d	E-conversion	0C	w	d
Dw.d	D-conversion	0E	w	d
Iw	I-conversion	10	w	
Tn	column set	12	n	
Aw	A-conversion	14	w	
Lw	L-conversion	16	w	
nX	skip or blank	18	n	
nHtext or 'text'	literal data	1A	n	text
)	group end	1C		
/	record end	1E		
Gw.d	G-conversion	20	w	d
	end of statement	22		
Zw	Hexadecimal conversion	24	w	

*The first hexadecimal bit of the byte indicates the scale factor sign (0 if positive, 1 if negative). The next seven bits contain the scale factor magnitude.

constant for the statement number associated with the FORMAT statement.

Processing Equivalence Entries: The STALL-IEKGST subroutine completes the processing of any equivalence entries in the information table that were not completed by prior routines in phase 10. These equivalence entries are the ones that were equivalenced before being dimensioned. The STALL-IEKGST subroutine computes displacements for each variable in the equivalence group.

Processing Literal Constants Used as Arguments: The STALL-IEKGST subroutine checks a pointer in the communication table (NPTR (1,27)) to see whether or not there are literal constants to process. If there are, the STALL-IEKGST subroutine calls IEKTLOAD and passes to it the location and length of the literal string that is used by the IEKTLOAD subroutine to generate literal text in the object module. All literal constants used as arguments are put on a double word boundary.

The STALL-IEKGST subroutine follows the chain in the literal constant dictionary entry and continues to call subroutine IEKTLOAD to process this text. After all the literal data text has been generated, the STALL-IEKGST subroutine adjusts the location counter by the amount of text generated. Literals used in DATA statements are not chained, and are not processed until CORAL is invoked.

Reserving Space for the Save Area: The STALL-IEKGST subroutine sets aside 76 bytes for the save area of the program being compiled.

Space in the object module for branch tables created by phase 10 for computed GO TO statements is also reserved by the STALL-IEKGST subroutine.

Checking for Undefined Statement Numbers: The STALL-IEKGST subroutine performs a dictionary scan for undefined statement numbers. This action is taken to ensure that every statement number that is referred to is also defined. The STALL-IEKGST subroutine scans the chain of statement number entries in the information table (see Appendix A: "Statement Number/Array Table") and examines a bit in the byte A usage field of each such entry. This bit is set by phase 10 to indicate whether or not it encountered a definition

of that statement number. If the bit indicates that the statement number is not defined, the STALL-IEKGST subroutine places an entry in the error table for later processing by phase 30.

Rechaining Entries for Variables: The STALL-IEKGST subroutine scans dictionary entries for variables. Previously executed routines in phase 10 sorted each variable chain alphabetically and left the pointer at the mid-item of the chain (for dictionary search speed). The STALL-IEKGST subroutine resets the pointer to the first (alphabetically lowest) item in the chain. The rechaining frees storage in each entry for later use by CORAL in phase 15. It then sets the adcon field of each dictionary entry for a variable to zero. The STALL-IEKGST subroutine also constructs dictionary entries for the imaginary parts of complex variables and constants.

Assigning Coordinates: The STALL-IEKGST subroutine calls subroutine IEKKOS which assigns coordinates to variables and constants in the following manner:

- The first 59 unique variables and/or constants that appear in the text created by phase 10 are assigned coordinates 2 through 60, respectively.[1] The coordinates are assigned in order of increasing coordinate number. (A coordinate between 2 and 60 may be assigned to a base variable if fewer than 59 unique variables and constants appear in the text.)

- The next 20 unique variables are assigned coordinates 61 through 80, respectively. The coordinates are assigned in order of increasing coordinate number. (If constants are encountered after coordinate 60 has been assigned, they are not assigned coordinates.)

- The coordinates 81 through 128 are reserved for assignment to base variables (see "Adcon and Base Variable Assignment" under "CORAL Processing").

Subroutine IEKKOS assigns to the first variable or constant in phase 10 text a coordinate number of 2, which indicates that the usage information for that variable or constant, regardless of the block in which it appears, is to be recorded in bit position 2 of the MVS, MVF, and MVX fields. The IEKKOS subroutine assigns to the second variable or constant a coordinate number of 3 and records its usage information in bit position 3 of the three fields. Subroutine IEKKOS continues this process until coordinate 60 has been assigned to a variable or constant. When coordinate number 60 has been assigned, the IEKKOS subroutine only assigns coordinates to the next 20 unique variables. Subroutine IEKKOS does not assign coordinates to or gather usage information for unique constants encountered after coordinate number 60 has been assigned. It assigns these variables coordinates 61 through 80, respectively. It records the usage information for each variable at the assigned bit location in the three fields. The IEKKOS subroutine does not assign coordinates to or gather usage information for unique variables encountered after coordinate number 80 has been assigned.

Subroutine IEKKOS uses a combination of the MCOORD vector, the MVD table, and the byte-C usage fields of the dictionary entries (see Appendix A, "Dictionary") to assign, keep track of, and record coordinate numbers. The MCOORD vector contains the number of the last coordinate assigned. The MVD table is composed of 128 entries, with each entry containing a pointer to the dictionary entry for the variable or constant to which the corresponding coordinate number is assigned or to the information table entry for the base variable to which the corresponding coordinate is assigned. The coordinate number assigned to a variable or constant is recorded in the byte-C usage field of the dictionary entry for that variable or constant.

Subroutine IEKKOS does not assign coordinates to or record usage information for unique constants encountered in text after coordinate number 60 has been assigned and unique variables encountered in text after coordinate number 80 has been assigned. If subroutine IEKKOS encounters a new constant after coordinate 60 has been assigned or a new variable after coordinate 80 has been assigned, it records a zero in the byte-C usage field of its associated dictionary entry. Phase 20 optimization deals only with those constants and variables that have been assigned coordinate numbers greater than or equal to 2 and less than or equal to 80.

Processing Common Entries in the Information Table: The STALL-IEKGST subroutine processes common entries in the information table. It computes the displacements of variables and arrays from

---

[1]The coordinate 1 is assigned to items such as unit numbers (i.e., data set reference numbers), complex variables in COMMON, arrays that are equivalenced, variables that are equivalenced to arrays, and variables that are equivalenced to variables of different modes.

the start of the common block that contains them and calculates the total size in bytes of each common block. Subroutine STALL-IEKGST records the displacements in the dictionary entries for the variables and the block size in the common table entry for the name of the common block. The displacements are used later to assign relative addresses to common variables. The block size is used by phase 25 to generate a control section for the common block (see Appendix A: "Common Table"). The STALL-IEKGST subroutine also places a pointer to the common table entry for the block name in the dictionary entry for each variable or array in that common block.

Processing Equivalence Entries in the Information Table: Subroutine STALL-IEKGST gathers additional information about equivalence groups and the variables in them. It computes a group head[1] and the displacement) of each variable in the group from this head. It records this information in the common table entries for the group and for the variables, respectively (see Appendix A: "Common Table"). Subroutine STALL-IEKGST identifies and flags in their dictionary entries variables and arrays put into common via the EQUIVALENCE statement. It also checks the variables and arrays for errors to verify that the associated common block has not been improperly extended because of the EQUIVALENCE declaration. If a common block is legitimately enlarged by an equivalence operation, the STALL-IEKGST subroutine recomputes the size of the common block and enters the size into the common table entry for the name of the common block.

If the name of a variable or array appears in more than one equivalence group, subroutine STALL-IEKGST recognizes the combination of groups and modifies the dictionary entries for the variables to indicate the equivalence operations. The STALL-IEKGST subroutine checks arrays that appear in more than one equivalence group to verify that conflicting relationships have not been established for the array elements.

During the processing of both common and equivalence information, a check is made to ensure that variables and arrays fall on boundaries appropriate to their defined types. If a variable or array is improperly aligned, subroutine STALL-IEKGST places an entry in the error table for processing by phase 30.

---

[1]The head of an equivalence group is that variable in the group from which all other variables or arrays in the group can be addressed by a positive displacement.

CONSTRUCTING A CROSS REFERENCE

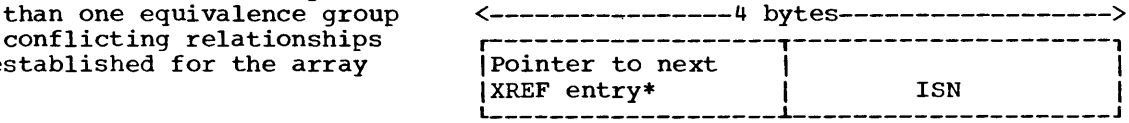
If the XREF option is selected, a two-part cross reference is constructed and written on the SYSPRINT data set immediately following the source listing. The first part of the cross reference is a list of all symbols used by the program and the ISNs of the statements in which each symbol appears. The symbols are written in alphabetic order and grouped by character length, first one-character symbols in alphabetic order, then two-character symbols in alphabetic order, etc. The second part of the cross reference is a sequential list of the statement numbers used on the program each followed by the ISN of the statement in which the statement number is defined and also by a list of the ISNs of statements that refer to the statement number.

XREF processing occurs during phase 10 and in a small separate overlay segment between phases 10 and 15. This segment, XREF-IEKXRF, is called only if the XREF option is selected.

Phase 10 Preparation for XREF Processing

If the XREF option is chosen, phase 10 subroutines LABTLU-IEKCLT and CSORN-IEKCCR perform additional processing for statement numbers and symbols. Also, phase 10 subroutine IEKXRS, which is not used unless the XREF option is chosen, is called.

The LABTLU-IEKCLT subroutine fills the adcon table, which is used as an XREF buffer, with XREF entries for statement number definitions and statement number references. The format of an XREF entry for statement numbers and symbols is:



* Relative to the beginning of the buffer.

Each time the buffer is full, the LABTLU-IEKCLT subroutine calls IEKXRS to write the buffer on SYSUT2. (The contents of SYSUT2 is later read in by subroutine XREF-IEKXRF and processed to produce a cross reference.) A count of the number of times the buffer is written out is kept in the communication table NPTR (2,20). Each time it finishes writing the buffer on SYSUT2, subroutine IEKXRS returns control to the LABTLU-IEKCLT subroutine.

Subroutine LABTLU-IEKCLT uses parts of the dictionary entries for statement numbers as pointers to keep track of its processing. It also adds a word (word 9) to each statement number dictionary entry to be used as a sequence chain field so that subroutine XREF-IEKXRF can create a sequential list of statement numbers used in the program.

The words used by the LABTLU-IEKCLT subroutine in dictionary entries for statement numbers are:

Word 5 - A pointer to the most recent statement number entry in the adcon table (XREF buffer) if the statement number reference being processed by subroutine LABTLU-IEKCLT is not a definition of a statement number. Word 5 is not used for statement number entries that correspond to definitions of statement numbers.

Word 6 - Bytes 1 and 2 -- The number of times the XREF buffer has been written on SYSUT2 at the time the statement number entry is processed by subroutine LABTLU-IEKCLT.

Bytes 3 and 4 -- A pointer to the first XREF buffer entry for the statement number.

Word 7 - Contains an ISN if the reference is to a definition of a statement number; contains -1 if the statement number has been previously defined.

Word 9 - Statement number sequence chain field.

The CSORN-IEKCCR subroutine processes symbols for XREF much the same way as subroutine LABTLU-IEKCLT processes statement numbers. However, for symbols, no processing is required for definitions and there is no sequencing.

The CSORN-IEKCCR subroutine adds one word to the dictionary entries for variables making a total of ten words in each entry. Word 10 for a variable entry is used in the same way as word 6 for a statement number entry. The first half of word 10 indicates the number of times the buffer has been written on SYSUT2 at the time the variable entry is processed by subroutine CSORN-IEKCCR. The second half of word 10 contains a pointer to the first XREF buffer entry for the symbol. The first half of word 8 is used as a pointer to the last (most recent) XREF buffer entry for the symbol.

Subroutine IEKXRS is also used during symbol processing to write the XREF buffer out on SYSUT2 whenever the buffer becomes full.

XREF Processing

If the XREF option is selected, the FSD calls the XREF-IEKXRF subroutine after the completion of subroutine STALL-IEKGST processing and before phase 15. The XREF-IEKXRF subroutine is a separate overlay segment that overlays phase 10 and is overlaid by phase 15.

Subroutine XREF-IEKXRF reads from SYSUT2 all buffers that were written out by IEKXRS during subroutine LABTLU-IEKCLT and subroutine CSORN-IEKCCR processing. It then sets up linkage between buffers for the symbol or statement number to create one sequential chain of ISNs and writes out the symbol or statement number with its ISNs on SYSPRINT. This process continues until all symbols and statement numbers with their ISNs are written on SYSPRINT. Control is then returned to the FSD that calls phase 15.

PHASE 15

Before phase 15 gains control, phase 10 has read the source statements, built the information table, and restructured the source statements into operator-operand pairs. When given control, phase 15 translates the text of arithmetic expressions, gathers information about branches and variables, converts phase 10 data text to a new text format, assigns relative addresses to constants and variables, and generates address constants when needed, to serve as address references. Thus, phase 15 modifies and adds to the information table and translates phase 10 normal and data text to their phase 15 formats.

Phase 15 is divided into two overlay segments, PHAZ15, and CORAL. Chart 05 shows the overall logic of the phase. Table 9 is a directory of all the subroutines used by phase 15.

PHAZ15 translates and reorders the text entries for arithmetic expressions from the operator-operand format of phase 10 to a four-part format suitable for phase 20 processing. The new order permits phase 25

to generate machine instructions in the correct sequence. PHAZ15 blocks the text and collects information describing the blocks. The information, needed during phase 20 optimization, includes tables on branching locations and on constant and variable usage.

CORAL, the second overlay segment of phase 15, performs a number of functions. It first converts phase 10 data text to a form more easily evaluated by subroutine DATOUT-IEKTDT. CORAL then assigns relative addresses to all variables, constants, and arrays. During one phase of relative address assignment, CORAL rechains phase 15 data text in order to simplify the generation of text card images by subroutine DATOUT-IEKTDT. CORAL also assigns address constants, when needed, to serve as address references for all operands.

PHAZ15 PROCESSING

The functions of PHAZ15 are text blocking, arithmetic translation, information gathering, and reordering of the statement number chain. Information gathering occurs only if optimization (either intermediate or complete) has been selected; it takes place concurrently with text blocking and arithmetic translation during the same scan of intermediate text. Reordering of the statement number chain occurs after PHAZ15 has completed the blocking, arithmetic translation, and information gathering.

PHAZ15 divides intermediate text into blocks for convenience in obtaining information from the text. Each block begins with a statement number definition and ends with the text entry just preceding the next statement number definition. An attempt is made to limit blocks to less than 80 text items as an aid to register routines in phase 20. PHAZ15 records information describing a text block in a statement number text entry and in an information table statement number entry.

During the same scan of text in which blocking occurs, PHAZ15 translates arithmetic expressions. The conversion is from the operation-operand pairs of phase 10 to a four-part format (phase 15 text). The new format follows the sequence in which algebraic operations are performed. In general, phase 15 text is in the same order in which phase 25 will generate machine instructions.[1] PHAZ15 copies, unchanged (except for rearrangement) into the text area, phase 10 text that does not require arithmetic translation or other special handling.

During the building of phase 15 text for a given block (if optimization has been selected), PHAZ15 constructs tables of information on the use of constants and variables in that text block. It stores information on variables and constants that are used within a block, and variables that are defined within a block. If complete optimization has been selected, PHAZ15 also gathers information on variables not first used and then defined. The foregoing usage information is recorded in the statement number text for each block for later use by phase 20.

Concurrently with text blocking, arithmetic translation, and gathering of constant/variable usage information, PHAZ15 discovers branching text entries and records the branching or connection information. This information, consisting initially of a table of branches from each text block (forward connections), is stored in a special array. Branching (connection) information is used during phase 20 optimization.

After PHAZ15 has completed the previously mentioned processing, it reorders the statement number chain of the information table. The original sequence of statement numbers, as phase 10 recorded them, was in the order of their occurrence in source statements as either definitions[2] or operands. Phase 15 reorders the statement numbers in the same sequence as they appeared as definitions in the source program. The new sequencing is established to facilitate phase 20 processing.

Last, PHAZ15 acquires a table of backward connection information consisting of branches into each statement number or text block. PHAZ15 derives this information from the forward connection information it previously obtained. Thus, connection information is of two types, forward and backward. PHAZ15 records a table of branches from each text block and a table of branches into each text block. Connection information of both types is used during phase 20 optimization.

---

[1] If optimization is selected, phase 20 may further manipulate the phase 15 text.

[2] A statement number occurs as a definition when that statement number appears to the left of a source statement.

Charts 06, 07, and 08 depict the flow of control during PHAZ15 execution. Table 10 lists the COMMON areas of phase 15.

Text Blocking

During its scan and conversion of phase 10 text, PHAZ15 sections the module into text blocks, which are the basic units upon which the optimization and register assignment processes of phase 20 operate. A text block is a series of text entries that begins with the text entry for a statement number and ends with the text entry that immediately precedes the text entry for thenext statement number. (The statement number may be either programmer defined or compiler generated.) When PHAZ15 encounters a statement number definition (i.e., the phase 10 text entry for a statement number), it begins a text block. It does this by constructing a statement number text entry (refer to Appendix B, "Phase 15 Intermediate Text Modifications"). PHAZ15 also places a pointer to the statement number text entry into the statement number entry (information table) for the associated statement number.

PHAZ15 resumes its scan and converts the phase 10 text entries following the statement number definition to their phase 15 formats. After each phase 15 text entry is formed and chained into text, PHAZ15 places a pointer to that text entry into the BLKEND field of the previously constructed statement number text entry. This field is, thereby, continually updated to point to the last phase 15 text entry.

When the next statement number definition is encountered, PHAZ15 begins the next text block in the previously described manner. A pointer to the text entry that ends the preceding block has already been recorded in the BLKEND field of the statement number text entry that begins that block. Thus, the boundaries of a text block are recorded in two places: the beginning of the block is recorded in the associated statement number entry (information table); the end of the block is recorded in the BLKEND field of the associated statement number text entry. All text blocks in the module are identified in this manner.

Note: For each ENTRY statement in the source module, phase 10 generates a statement number text entry and places it into text preceding the text for the ENTRY statement. Phase 10 also ensures that the statement following an ENTRY statement has a statement number; if a statement number is not provided by the programmer, phase 10 generates one. Thus, the text entries for each ENTRY statement form a separate text block, which is referred to as an entry block.

Figure 3 illustrates the concept of text blocking. In the illustration, two text blocks are shown: one beginning with statement number 10; the other with statement number 20. The statement number entry for statement number 10 contains a pointer to the statement number text entry for statement number 10, which contains a pointer to the text entry that immediately precedes the statement number text entry for statement number 20. Similar pointers exist for the text block starting with statement number 20.

Arithmetic Translation

Arithmetic translation is the reordering of arithmetic expressions in phase 10 text format to agree with the sequence in which algebraic operations are performed. Arithmetic expressions may exist in IF, CALL, and ASSIGN statements and input/output data-lists, as well as in arithmetic statements and statement functions.

When PHAZ15 detects a primary adjective code for a statement that needs arithmetic translation, it passes control to the arithmetic translator (ALTRAN-IEKJAL). If the phase 10 text for the statement does not require any type of special handling, ALTRAN-IEKJAL reorders it into a series of phase 15 text entries that reflect the sequence in which arithmetic operations are to be carried out. During the reordering process, ALTRAN-IEKJAL calls various supporting routines that perform checking and resolution (e.g., the resolution of operations involving operands of different modes) functions.

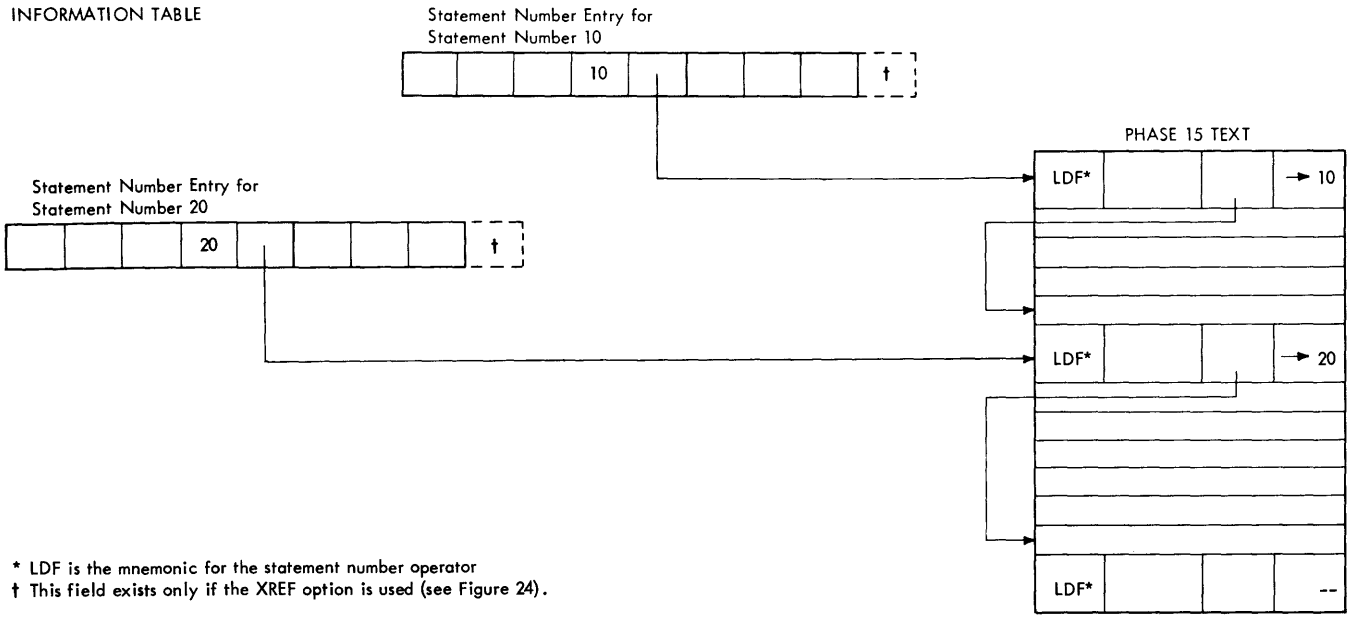


Figure 3. Text Blocking

Throughout the reordering process, ALTRAN-IEKJAL is checking for text that requires special handling before it can be placed into the phase 15 text area. [Special handling is required for complex expressions, terms involving unary minuses (e.g., A=-B), subscript expressions, statement function references, etc.] If special text processing is required, ALTRAN-IEKJAL calls one or more subroutines to perform the required processing.

During reordering and, if required, special handling, subroutine GENER-IEKLGN is called to format the phase 15 text entries and to place them into the text area.

REORDERING ARITHMETIC EXPRESSIONS: The reordering of arithmetic expressions is done by means of a pushdown table. This table is a last-in, first-out list. After the table is initialized (i.e., the first operator-operand pair of an arithmetic expression is placed into the table), the arithmetic translator (ALTRAN-IEKJAL) compares the operator of the next operator-operand pair (term) in text with the operator of the pair at the top of the pushdown table. As a result of each comparison, either a term is transferred from phase 10 text to the table, or an operator and two operands (triplet) are brought from the table to the phase 15 text area, eliminating the top term in the pushdown table.

The comparison made to determine whether a term is to be placed into the pushdown table or whether a triplet is to be taken from the pushdown table is always between the operator of a term in phase 10 text and the operator of the top term in the table. Each comparison is made on the basis of relative forcing strength. A forcing strength is a value assigned to an operator that determines when that operator and its associated operands are to be placed in phase 15 text. The relative values of forcing strengths reflect the hierarchy of algebraic operations. The forcing strengths for the various operators appear in Table 2.

When the arithmetic translator (ALTRAN-IEKJAL) encounters the first operator-operand pair (phase 10 text entry) of a statement, the pushdown table is empty. Since the translator cannot yet make a comparison between text entry and table element, it enters the first text entry in the top position of the table. The translator then compares the forcing strength of the operator of the next text entry with that of the table element. If the strength of the text operator is greater than that of the top (and only) table element, the text entry (operator-operand pair) becomes the top element of the table. The original top element is effectively "pushed down" to the next lower position. In Figure 4, the number-1 section of the drawing shows the pushdown table at this time.

The operator of the next text entry (operator C--operand C at section 2) is compared with the top table element

(operator B--operand B at section 1) in a similar manner.

Table 2. Operators and Forcing Strengths

Operator	Forcing Strength
End Mark	1
=	2
)	3
,	6
.OR.	7
.AND.	8
.NOT.	9
.EQ., .NE., .GT., .LT., .GE., .LE.	10
+, -, minus(	11
*, /	12
**	13
(f -- left parenthesis after a function name	14
(s -- left parenthesis after an array name	15
(	16

When a comparison of forcing strengths indicates that the strength of the text operator (operator C, section 2), is less than or equal to that of the top table element (operator B), the table element is said to be "forced." The forced operator (operator B) is placed in the new phase 15 text entry (section 3 of the illustration) with its operand (operand B) and the operand of the next lower table entry (operand A). Note that subroutine ALTRAN-IEKJAL has generated a new operand t (see section 3) called a "temporary." A temporary is a compiler-generated operand in which a preliminary result may be held during object-module execution.[1] With operator B, operand B, and operand A (a triplet) removed from the pushdown table, the previously entered operator-operand pair (operator A, section 1) now becomes the top element of the table (section 4). The ALTRAN-IEKJAL subroutine assigns the previously generated temporary t as the operand of this pair. This temporary represents the previous operation (operator B--operand A--operand B).

---

[1]A given temporary may be eliminated by phase 20 during optimization.

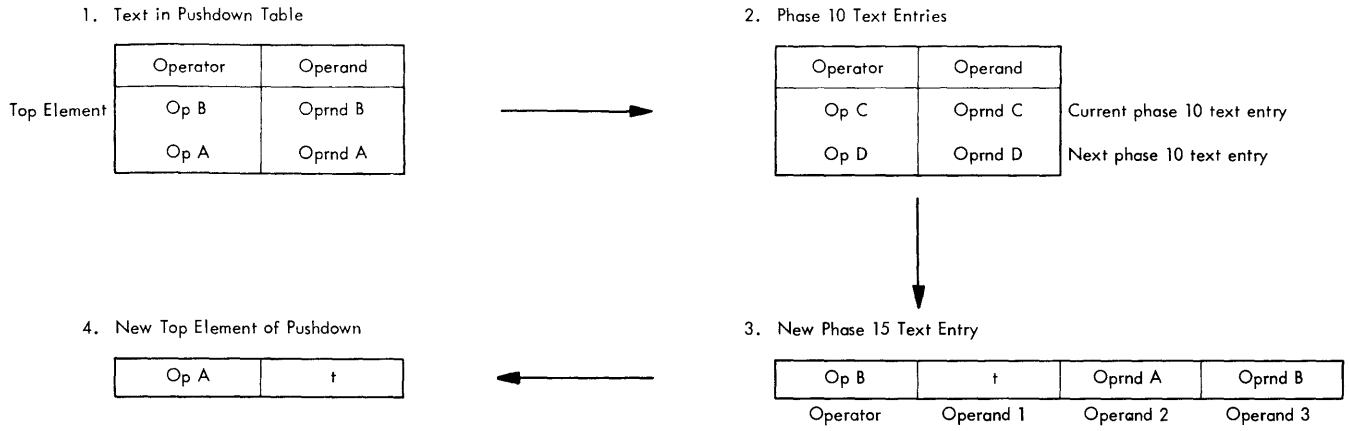
Comparisons and text-to-table exchanges continue, a higher strength text operator "pushing" a phase 10 text entry into the table and a lower strength text operator "forcing" the top table operator and its operands (triplet) from the table. In each case, the forced table items become the new phase 15 text entry. An exception to the general rule is a left parenthesis, which has the highest forcing strength. Operators following the left parenthesis can be forced from the table only by a right parenthesis, although the intervening operators (between the parentheses) are of lower forcing value. When the translator reaches an end mark in text, its forcing strength of 1 forces all remaining elements from the table.

SPECIAL PROCESSING OF ARITHMETIC EXPRESSIONS: As stated before, arithmetic translation involves reordering a group of phase 10 text entries to produce a new group of phase 15 text entries representing the same source statement. Certain types of entries, however, need special handling (for example, subscripts and functions). When it has been determined that special handling is needed, control is passed to one or more other subroutines (see Chart 07) that perform the desired processing.

The following expressions and terms need special handling before they are placed in phase 15 text: complex expressions, terms involving a unary minus, terms involving exponentiation, commutative expressions, subscript expressions, subroutine or function subprogram references, statement function references, and expressions involved in logical IF statements.

Complex Expressions: A complex expression is converted into two expressions, a real expression and an imaginary one. For real elements in the expression, complex temporaries are generated with zero in the imaginary part and the real element in the real part. For example, the complex expression B + C + 25. is treated as:

B real	+	C real	+	25. real
B imag	+	C imag	+	0. imag



NOTE: A phase 15 text entry having an arithmetic operator may be envisioned as operand 1 = operand 2 - operator - operand 3, where the equal sign is implied.

Figure 4. Text Reordering via the Pushdown Table

An expression is not treated as complex if the "result" operand (left of the equal sign in the source statement) is real. In this case, the translator places only the real part of the expression in phase 15 text. But if a complex multiplication, division, or exponentiation is involved in the expression, the real and imaginary parts will appear in phase 15 text, but only the real part of the result will be used at execution time.

Terms Containing a Unary Minus: In terms that contain unary minuses, the unary minuses are combined with additive operators (+, -) to reduce the number of operators. This combining, done by subroutine UNARY-IEKKUN, may result in reversed operators or operands or both in phase 15 text. For example, -(B-C) becomes C-B, and A+(-B) becomes A-B. This process reduces the number of machine instructions that phase 25 must generate.

Operations Involving Powers: Several kinds of special handling are provided by subroutine UNARY-IEKKUN for operations involving powers. Multiplications by powers of two are converted to left shift operations. A constant integer power of two raised to a constant integer power is converted to the equivalent left shift operation. Last, a constant or variable raised to a constant integer power is converted to a series of multiplications (and a division operation into 1, if the power is negative). This conversion is a function of the level of optimization selected. This handling requires less execution time than using an exponentiation subroutine.

Commutative Operations: If an operation is commutative (either operand can be operated upon, such as in adding or multiplying), the two operands are reordered to agree with their absolute locations in the dictionary.

Subscripts: Subroutines SUBMULT-IEKKSM and SUBADD-IEKKSA perform subscript processing. Subscripted items are processed one at a time throughout the subscript. If the subscript itself is an expression, it is first processed via the translator. Text entries are then generated to multiply the subscript variable by the dimension factor and length. Each subscript item is handled in a similar manner. When all subscript items have been processed, phase 15 text entries are generated to add all subscript values together to produce a single subscript value.

In general, during compilation, constants in subscript expressions are combined, and their composite value is placed in the displacement field of the phase 15 text entry for the subscript item (see Appendix B, "Phase 15/Phase 20 Intermediate Text Modifications"). Phase 25 uses the value in the displacement field to generate, in the resultant object instructions, the displacement for referring to the elements in the array. This combining of constants reduces the number of instructions needed during execution to compute the subscript value.

Expressions Referring to In-Line Routines or Subprograms: Expressions containing references to in-line routines or subprograms are processed by the following subroutines: FUNDRY-IEKJFU, BLTNFN-IEKJBF, and DFUNCT-IEKJDF.

Arguments that are expressions are reduced by the translator to a single temporary, which is used as the argument. If an argument is a subscripted variable, subscript processing (previously discussed) reduces the subscript to a single subscripted item. Either subroutine DFUNCT-IEKJDF (for references to library routines) or subroutine BLTNFN-IEKJBF (for references to in-line routines) then conducts a series of tests on the argument and performs the processing determined by the results of the tests.

If a function is not external and is in the function table (IEKLFT) (see Appendix A, "Function Table"), it is determined if the required routine is in-line. If the function is in-line and its mode (or the mode of its arguments) is not as expected, it is assumed that the function is external. If there are no error conditions, subroutine BLTNFN-IEKJBF either generates text or substitutes a special operator (such as those for ABS or FLOAT) in the phase 15 text so that phase 25 can later expand the function. Phase 15 provides some in-line routines itself.[1] Instead of placing a special operator in text, phase 15 inserts a regular operator, such as the operator for AND or STORE.

If the mode of arguments in a library function is not as expected, another test is performed. The test determines whether or not a previous reference was made correctly for these arguments. If the previous reference was as expected, it is assumed that an error exists. Otherwise, the function is assumed to be external.

If a function is assumed to be external (either used in an EXTERNAL statement or does not appear in the function table), text is generated to load the addresses of any arguments that are subscripted variables into a parameter list. (If none of the arguments are subscripted variables, the load address items are not required.) A text entry for a subroutine or a function call is then generated. The operator of the text entry is for an external function or subroutine reference. The entry points to the dictionary entry for the name. The text representation of the argument list is then generated and placed into the phase 15 text chain.

---

[1]BLTNFN-IEKJBF expands the following functions: TBIT, SNGL, REAL, AIMAG, DCMPLX, DCONJG, and CONJG.

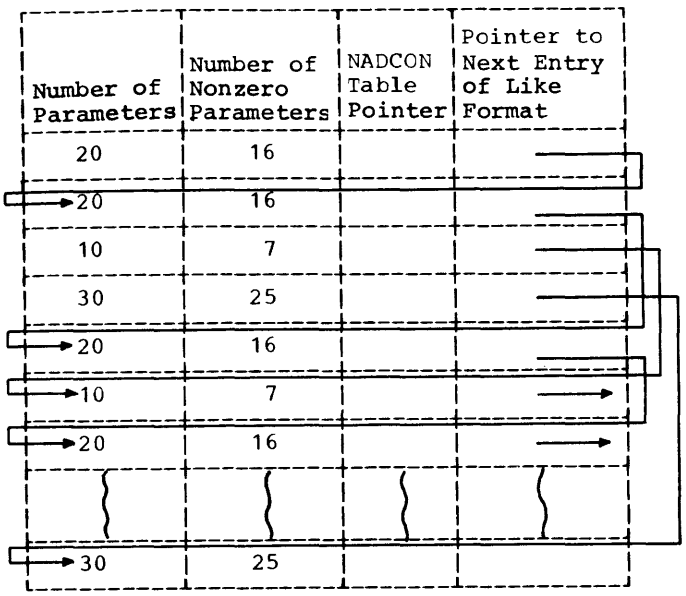
If a function is in the function table, but does not represent an in-line routine, text is generated to load the addresses of any arguments that are subscripted variables into a parameter list. (Load address items are not required if none of the arguments are subscripted variables.) A text entry having a library function operator is generated. This entry points to the dictionary entry for the function. The text representation of the argument list is then generated and placed into the phase 15 text chain.

Parameter List Optimization: Subroutine DFUNCT-IEKJDF performs parameter list optimization. If two or more parameter lists are identical, all but one can be eliminated. Likely candidates for optimization are those parameter lists with (1) the same number of parameters and (2) the same nonzero parameters. When two such lists are found, individual parameters are compared to determine whether the lists are actually identical or merely of the same format.

To make the comparison easier, the Parameter List Optimization Table is formed. Its format is:

Number of parameters in list	Number of nonzero parameters in list	Pointer to NADCON table entry	Pointer to next entry of like format in this table
1 byte	1 byte	1 byte	1 byte

For each unique parameter list, an entry is made in the table describing the number of parameters in the list, the number of non zero parameters in the list, a pointer to the adcon table (see Appendix A: "NADCON Table") and a pointer to the next parameter list optimization table entry that contains a like parameter list format, but unlike individual parameters. When a new parameter list is generated, the parameter list optimization table is scanned for a possible identical list. If one is found, the parameters in the new list are compared with the parameters in the old list. If the lists are identical, a pointer to the old list is used as the new list's pointer. If the lists are not identical, an entry for the new list is made in the table and chained to the last like (in format) entry. For example:



Parameter list optimization is limited to (1) 100 entries in the parameter list optimization table or (2) 255 entries in the adcon table. No further parameter list optimization is attempted if either limit is exceeded.

Expressions Containing Statement Function References: For expressions containing statement function references, the arguments of the statement function text are reduced to single operands (if necessary). These arguments and their mode are stored in an argument save table (NARGSV), which serves as a dictionary for the statement function skeleton pointed to by the dictionary entry for the statement function name. The argument save table is used in conjunction with the usual pushdown procedure to generate phase 15 text items for the statement function reference. When the translator encounters an operand that is a dummy argument, the actual argument corresponding to the dummy is picked up from the argument save table and replaces the dummy argument.

Logical Expressions: Subroutines ALTRAN-IEKJAL, ANDOR-IEKJAN, and RELOPS-IEKKRE perform a special process, called anchor point, on logical expressions containing relational operators, ANDs, ORs, and NOTs, so that, at object time, unnecessary logical tests are eliminated. With anchor-point "optimization," only the minimum number of object-time logical tests are made before a branch or fall-through occurs. For example, with anchor-point handling, the statement IF(A.AND.B.AND.C) GO TO 500 will produce (at object time) a branch to the next statement if A is false, because B and C need not be tested. Thus, only a minimum number of operands will be tested. Without anchor-point handling of the expression during compilation, all operands would be tested at object time. Similar special handling occurs for text containing logical ORs.

When a primary adjective code for a logical IF statement or an end-of-DO IF is placed in the pushdown table, a scan of phase 10 text determines whether or not the associated statement can receive anchor-point handling. The statement can receive anchor-point handling if two conditions are met. There must not be a mixture of ANDs and ORs in the statement. A logical expression, if it is in parentheses, must not be negated by the NOT operator. If these two conditions are not met, special handling of the logical expression does not occur.

Gathering Constant/Variable Usage Information

During the conversion of the phase 10 text entries that follow the beginning of a text block (i.e., the text entries that follow a statement number definition) to phase 15 format, the PHAZ15 subroutine MATE-IEKLMA gathers usage information for the variables and constants in that block. This information is required during the processing of the optimizer path through phase 20 (see "Phase 20"). If optimizer processing is not selected, this information is not compiled. Subroutine MATE-IEKLMA records the usage information in three fields (MVS, MVF, and MVX), each 128 bits long, of the statement number text entry for the block (see Appendix B, "Phase 15 Intermediate Text Modifications"). The MVS field indicates which variables are defined (i.e., appear in the operand 1 position of a text entry) within the text of the block. The MVF field indicates which variables, constants, and base variables (see "Adcon and Base Variable Assignment" under "CORAL Processing") are used (i.e., appear in either the operand 2 or operand 3 position of a text entry) within the text of the block. The MVX field indicates which variables are defined but not first used (not busy-on-entry) within the text of the block. The MVX information is gathered for the second level of optimization only.

Subroutine MATE-IEKLMA records the usage information for a variable or constant at a specific bit location within the three fields. (Base variables are processed during CORAL processing.) The bit location at which the usage information is recorded is determined from the coordinate assigned to the variable or constant by subroutine IEKKOS.

After a phase 15 text entry has been formed, subroutine MATE-IEKLMA is given control to determine and record the usage information for the text entry. It examines the text entry operands in the order: operand 2, operand 3, operand 1. If operand 2 has not been assigned a coordinate, subroutine MATE-IEKLMA assigns it the next coordinate, enters the coordinate number into the dictionary entry for the operand, and places a pointer to that dictionary entry into the MVD table entry associated with the assigned coordinate number. After MATE-IEKLMA has assigned the coordinate, or if the operand was previously assigned a coordinate, it records the usage information for the operand. The operand's associated coordinate bit in the MVF field (of the statement number text entry for the block containing the text entry under consideration) is set to on, indicating that the operand is used in the block. Subroutine MATE-IEKLMA executes a similar procedure to process operand 3 of the text entry.

If operand 1 of the text entry has not been assigned a coordinate, the MATE-IEKLMA subroutine assigns the next coordinate to it and records the following usage information for operand 1:

- Its associated coordinate bit in the MVX field is set to on only if the associated coordinate bit in the MVF field is not on. (If the associated MVF bit is on, operand 1 of the text entry was previously used in the block and, therefore, is not not busy-on-entry.)
- Its associated coordinate bit in the MVS field is set to on, indicating that it is defined within the block.

This process is repeated for all of the phase 15 text entries that are formed following the construction of a statement number text entry and preceding the construction of the next statement number text entry. When the next statement number text entry is constructed, all of the usage information for the preceding block has been recorded in the statement number text entry that begins that block. The same procedure is followed to gather the usage information for the next text block.

Gathering Forward-Connection Information

An integral part of the processing of PHAZ15 is the gathering of forward-connection information, which indicates the specific text blocks that pass control to other specified text blocks. Forward-connection information is used during phase 20 optimization.

Forward-connection information is recorded in a table called RMAJOR. Each RMAJOR entry is a pointer to the statement number entry associated with a statement number that is the object of a branch or a fall-through. Because each statement number entry contains a pointer to the text block beginning with its associated statement number (see "Text Blocking"), each RMAJOR entry points indirectly to a text block.

For each new text block, PHAZ15 places a pointer to the next available entry in RMAJOR into the forward-connection field of the associated statement number entry (see Appendix A, "Statement Number/Array Table"). Thus, the statement number entry associated with the text block points to the first entry in RMAJOR in which the forward-connection information for that block is to be recorded.

After starting a text block, PHAZ15 converts the phase 10 text following the statement number definition to phase 15 text. As each phase 15 text entry is formed, it is analyzed to determine whether it is a GO TO or compiler generated branch. If it is either, a pointer to the statement number entry for each statement number to which a branch may be made as a result of the execution of the GO TO or generated branch is recorded in the next available entry in RMAJOR. (If two or more branches to the same statement number appear in the block only one entry is made in RMAJOR for the statement number to which a branch is to be made.)

When PHAZ15 encounters the next statement number definition, it starts a new block. If the new block is an entry block, PHAZ15 saves a pointer to its associated statement number entry for subsequent use and processes the text for the block.

If the new block is neither an entry block nor an entry point (i.e., a block immediately following an entry block), PHAZ15 records the fall-through connection

information (if any) for the previous block. If the previous block is terminated by an unconditional branch, it does not fall-through to the new block. If the previous block can fall-through to the new block, PHAZ15 records a pointer to the statement number entry for the new block in the next location of RMAJOR. It then flags this as the last forward connection for the previous block.

If the new block is an entry point (i.e., a block immediately following an entry block), PHAZ15 records the fall-through connection (if any) for the previous non-entry block. It does this in the manner described in the previous paragraph. It then records the forward-connection information for all intervening entry blocks (i.e., entry blocks between the previous non-entry block and the new block). (PHAZ15 has saved pointers to the statement number entries for all intervening entry blocks.) Each such entry block passes control directly to the new block and therefore has only one forward connection. To record the forward connection information for the intervening entry blocks, PHAZ15 places a pointer to the next available entry in RMAJOR into the forward connection field of the statement number entry for the first intervening entry block. In this RMAJOR entry, PHAZ15 records a pointer to the statement number entry for the new block. It flags this entry as the last, and only, RMAJOR entry for the entry block. PHAZ15 repeats this procedure for the remaining intervening entry blocks (if any). PHAZ15 then proceeds to process the new text block.

When all the connection information for a block has been gathered, each RMAJOR entry for the block, the first of which is pointed to by the statement number entry for the block and the last of which is flagged as such, points indirectly to a block to which that block may pass control.

Figure 5 illustrates the end result of gathering forward-connection information for sample text blocks. Only the forward-connection information for the blocks beginning with statement numbers 10 and 20 is shown. In the illustration, it is assumed that:

- The block started by statement number 10 may branch to the blocks started by statement numbers 30 and 40 and will fall-through to the block started by statement number 20 if neither of the branches is executed.
- The block started by statement number 20 may branch to the blocks started by statement numbers 40 and 50 and will fall-through to the block started by statement number 30 if neither of the branches is executed.

Reordering the Statement Number Chain

After text blocking, arithmetic translation, and if complete optimization has been specified, the gathering of constant/variable usage information, been completed, subroutine PHAZ15-IEKJA reorders the statement number chain of the information table (see Appendix A, "Information Table"). The original sequence of the entries in this chain, as recorded by phase 10, was in the order of the occurrence of their associated statement numbers as either definitions or operands. The new sequence of the entries after reordering is made according to the occurrence of their associated statement numbers as definitions only.

Although the actual reordering takes place after the scan of the phase 10 text, preparation for it takes place during the scan. As each statement number definition is encountered, a pointer to the related statement number entry is recorded. Thus, during the course of processing, a table of pointers to statement number entries, which reflects the sequence in which statement numbers are defined in the module, is built. The order of the entries in this table also reflects the sequence of the text blocks of the module.

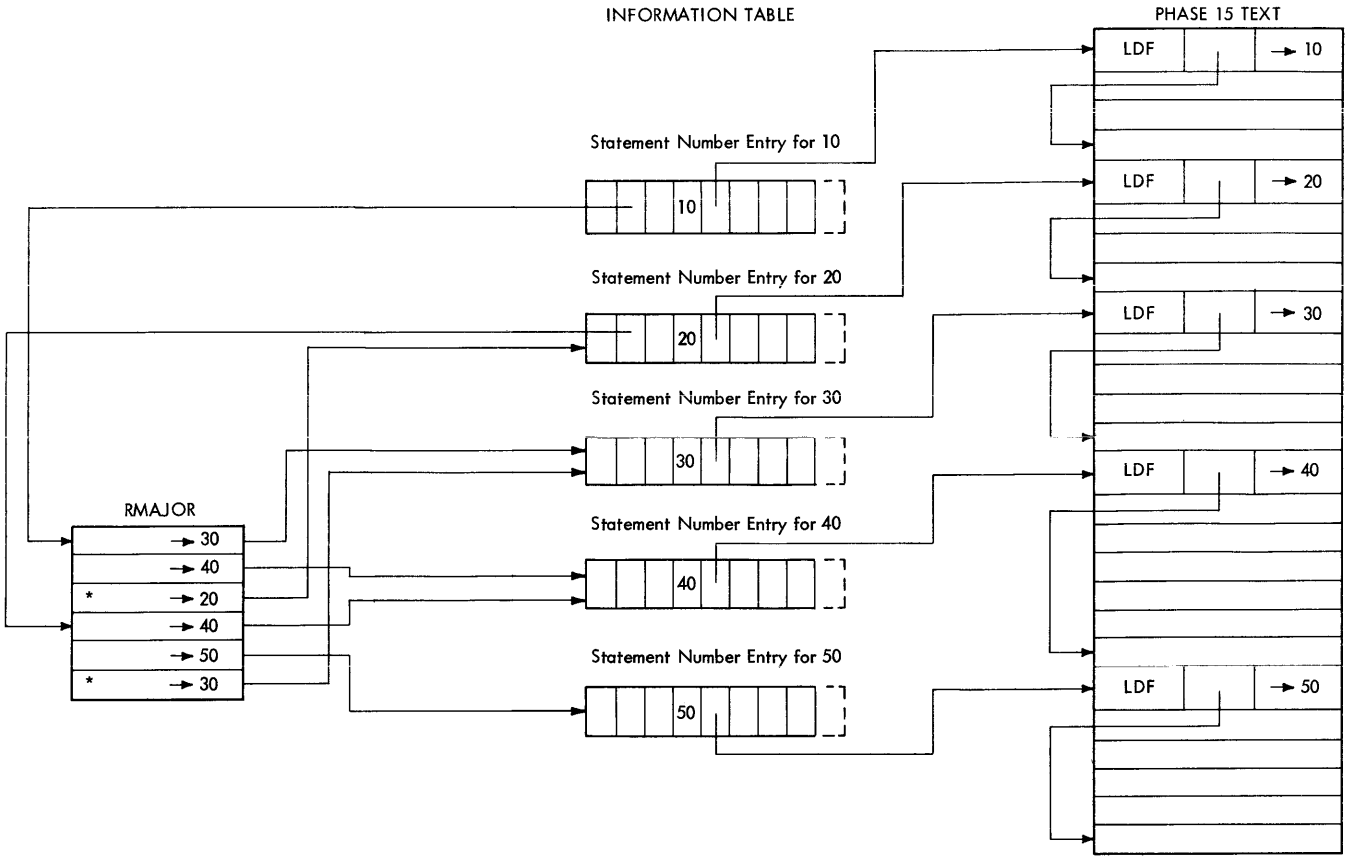


Figure 5. Forward-Connection Information

After the scan, subroutine PHAZ15-IEKJA uses this table to reorder the statement number entries. It places the first table pointer into the appropriate field of the communication table (see Appendix A, "Communication Table"); it places the second table pointer into the chain field of the statement number entry that is pointed to by the pointer in the communication table; it places the third table pointer into the chain field of the statement number entry that is pointed to by the chain field of the statement number entry that is pointed to by the pointer in the communication table; etc. When subroutine PHAZ15-IEKJA has performed this process for all pointers in the table, the entries in the statement number chain are arranged in the sequence in which their associated statement numbers are defined in the module. The new order of the chain also reflects the sequence of the text blocks of the module.

Gathering Backward-Connection Information

After the statement number chain has been reordered, and if optimization has been specified, subroutine PHAZ15-IEKJA gathers backward-connection information. This information indicates the specified text blocks that receive control from specific other text blocks. Backward-connection information is used extensively throughout phase 20 optimization.

Subroutine PHAZ15-IEKJA uses the reordered statement number chain and the information in the forward connection table (RMAJOR) to determine the backward connections. It records backward-connection information in a table called CMAJOR in subroutine C1520-IEKJA2. Each CMAJOR entry made by subroutine PHAZ15-IEKJA for a particular text block (block I) is a pointer to the statement number entry for a block from which block I may receive control. Because each statement number entry contains a pointer to its associated text block (see "Text Blocking"), each CMAJOR entry for block I points indirectly to a block from which block I may receive control.

Subroutine PHAZ15-IEKJA gathers backward-connection information for the text blocks according to the order of the statement number chain. It first

determines and records the backward-connections for the text block associated with the initial entry in the statement number chain, then gathers the backward-connection information for the block associated with the second entry in the chain; etc.

For each text block, subroutine PHAZ15-IEKJA initially records a pointer to the next available entry in CMAJOR in the backward-connection field (JLEAD) of the associated statement number entry (see Appendix A, "Statement Number/Array Table"). Thus, the statement number entry points to the first entry in CMAJOR in which the backward-connection information for the block is to be recorded.

Then, to determine the backward-connection information for the block (block I), subroutine PHAZ15-IEKJA obtains, in turn, each entry in the statement number chain. (The entries are obtained in the sequence in which they are chained.) After the PHAZ15-IEKJA subroutine has obtained an entry, it picks up the forward-connection field (ILEAD) of that entry. This field points to the initial RMAJOR entry for the text block associated with the obtained statement number entry. (Note: RMAJOR entries for a block indicate the blocks to which that block may pass control.) Subroutine PHAZ15-IEKJA searches all RMAJOR entries for the block associated with the obtained entry for a pointer to the statement number entry for block I. If such a pointer exists, the text block associated with the obtained statement number entry may pass control to block I. Therefore, block I may receive control from that block and subroutine PHAZ15-IEKJA records a pointer to its associated statement number entry in the next available entry in CMAJOR. Subroutine PHAZ15-IEKJA repeats this procedure for each entry in the statement number chain. Thus, it searches all RMAJOR entries for pointers to the statement number entry for block I and records in CMAJOR a pointer to the statement number entry for each text block from which block I may receive control. The PHAZ15-IEKJA subroutine flags the last entry in CMAJOR for block I. When the statement number chain has been completely searched, subroutine PHAZ15-IEKJA has gathered all the backward-connection information for block I. Each entry that the PHAZ15-IEKJA subroutine has made for block I, the first of which is pointed to by the statement number entry for block I and the last of which is flagged, points indirectly to a block from which block I may receive control.

Subroutine PHAZ15-IEKJA gathers the backward-connection information for all blocks in the aforementioned manner. When all of this information has been gathered, control is returned to the FSD, which calls CORAL, the second segment of phase 15.

Figure 6 illustrates the end result of the gathering of backward-connection information for sample text blocks. Only the backward-connections for the blocks beginning with statement numbers 40 and 50 are shown. In the illustration, it is assumed that:

- The block started by statement number 40 may receive control from the execution of branch instructions that reside in the blocks started by statement numbers 10 and 20 and that it may receive control as a result of a fall-through from the block started by statement number 30.

- The block started by statement number 50 may receive control from the execution of a branch instruction that resides in the block started by statement number 20 and that it may receive control as a result of a fall-through from the block started by statement number 40.

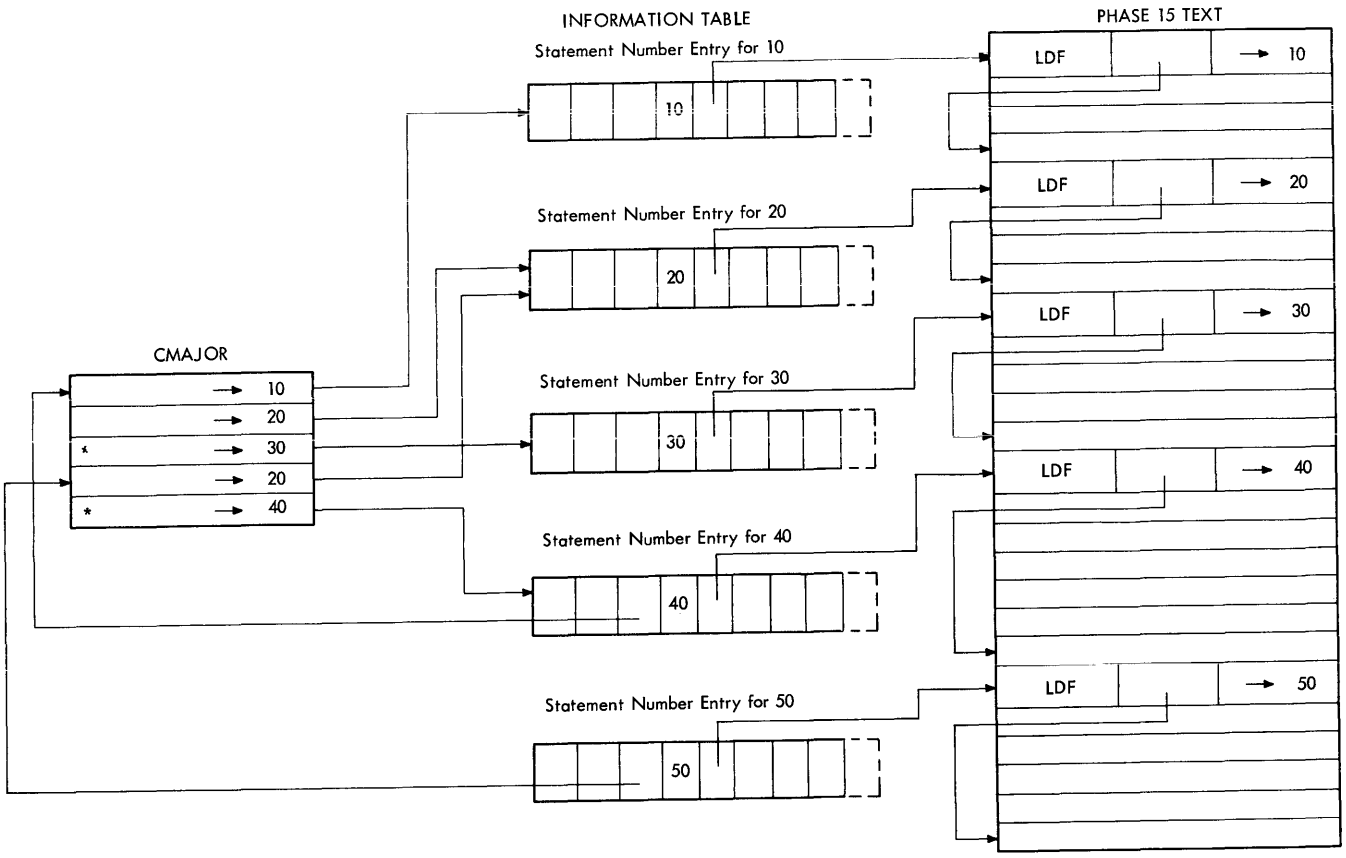


Figure 6. Backward-Connection Information

CORAL PROCESSING

CORAL, the second segment of phase 15, performs the following functions:

- Data text conversion
- Relative address assignment
- Data text rechaining
- Namelist statement processing
- Define file text processing
- Initial value assignment
- Adcon table space reservation

CORAL consists of a main subroutine, CORAL-IEKGCR, which controls the flow of space allocation for variables, constants, and any adcons necessary for local variables, COMMON, EQUIVALENCE, and EXTERNAL references. Embedded in subroutine CORAL-IEKGCR are the routines that process constants, local variables, and external references. The CORAL-IEKGCR subroutine calls other routines in phase 15 to accomplish various functions. These routines are:

- IEKGCZ, which keeps track of space being allocated; generates adcons needed for address computation in the object module; rechains data text in the sequence of variable assignment; generates adcons necessary for COMMON, EQUIVALENCE, and EXTERNAL references; and sets up error table entries to be used by phase 30 if errors occur.
- NDATA-IEKGDA, which processes phase 10 data text.
- EQVAR-IEKGEV, which handles COMMON and EQUIVALENCE space allocation.
- NLIST-IEKTNL, which processes namelist text.
- DFILE-IEKTDF, which processes define file text.
- DATOUT-IEKTDT, which processes data text.

Chart 09 shows the overall logic flow of CORAL.

Translation of Data Text

The first section of CORAL, subroutine NDATA-IEKGDA, translates data text entries from their phase 10 format to a form more easily processed by another CORAL subroutine, DATOUT-IEKTDT. Each phase 10 data text entry (except for initial housekeeping entries) contains a pointer to a variable or constant in the information table. Each variable in the series of entries is to be assigned to a constant appearing in another entry. Placed in separate entries, variable and constant appear to be unrelated. In each phase 15 data text entry, after translation, each related variable and constant are paired (they appear in adjacent fields of the same entry).

The following example shows how a series of phase 10 data text entries are translated by the NDATA-IEKGDA subroutine to yield a smaller number of phase 15 text entries, with each related constant and variable paired. Assume a statement appearing in the source module as DATA A,B/2*0/. The resulting phase 10 text entries appear as follows (ignoring the chain, mode, and type fields, and the initial housekeeping entry):

Adjective Code for:	Pointer
0	Pointer to A in dictionary
,	Pointer to B in dictionary
/	2
*	Pointer to 0 in dictionary
/	0

Note that the variables A and B and the constant value 0 appear in separate text entries. The NDATA-IEKGDA subroutine translation of the above phase 10 entries (ignoring the contents of the indicator and chain fields, and two optional fields needed for special cases) appears as follows:

Indicator	Chain	P1 Field	P2 Field
		pointer to A in dictionary	pointer to 0 in dictionary
		pointer to B in dictionary	pointer to 0 in dictionary

In this case, each variable and its specified constant value appear in adjacent fields of the same phase 15 text entry. For the detailed format of the phase 15 data text entry and the use of the special fields not discussed, see Appendix B, "Phase 15/20 Intermediate Text Modification".

Relative Address Assignment

The chief function of CORAL is to assign relative addresses to the operands (constants and variables) of the source module. The addresses indicate the locations, relative to zero, at which the operands will reside in the object module resulting from the compilation. The relative address assigned to an operand consists of an address constant and a displacement. These two elements, when added together, form the relative address of the operand. The address constant for an operand is the base address value used to refer to that operand in main storage. Address constants are recorded in the adcon table (NADCON) and are the elements to which the relocation factor is added to relocate the object module for execution. The displacement for an operand indicates the number of bytes that the operand is displaced from its associated address constant. Displacements are in the range of 0 to 4095 bytes. The relative address assigned to an operand is recorded in the information table entry for that operand in the form of:

1. A numeric displacement from its associated address constant.

2. A pointer to an information table entry that contains a pointer to the associated address constant in the adcon table.

Relative addresses are assigned through use of a location counter. This counter is continually updated by the size (in bytes) of the operand to which an address is assigned. The value of the location counter is used to:

- Compute the displacement to be assigned to the next operand.
- Determine when the next address constant is to be established. (If the displacement reaches a value in excess of 4095, a new address constant is established.)

CORAL assigns addresses to source module operands in the following order:

- Constants.
- Variables.
- Arrays.
- Equivalenced variables and arrays.
- COMMON variables and arrays, including variables and arrays made common using the EQUIVALENCE statement.

The manner in which addresses are assigned to each of these operand types is described in the following paragraphs. Because constants and variables are processed in the same manner, they are described together.

Constants and Variables: Subroutine CORAL-IEKGCR first assigns relative addresses to the constants of the module. As each constant is assigned a relative address, subroutine CORAL-IEKGCR calls the FSD subroutine, IEKTLOAD, to place the constant in the object module in the form of TXT records. Addresses are then assigned to variables. (In the subsequent discussion, constants and variables are referred to collectively as operands.) The first operand is assigned a displacement of zero plus the length of the save area, parameter list, and branch table. Operands that are assigned locations within the first 4096 bytes of the range of base register 13 are not explicitly assigned an address constant. Such operands use the base address value loaded into reserved register 13 as their address constant. The displacement is recorded in the information table entry for that operand. The location counter is then updated by the size in bytes of the operand.

The next operand is assigned a displacement equal to the current value of the location counter minus the base address value in register 13. The displacement is recorded in the information table entry for that operand. The location counter is then updated, and the value of the displacement is tested to see whether or not it exceeds 4095. If it does not, the next operand is processed as described above.

If sufficient operands exist to cause the displacement to achieve a value in excess of 4095, the first address constant is established. The value of this address constant equals the location counter value that caused its establishment. This address constant becomes the current address constant and is saved for subsequently assigned relative addresses. The displacement value is then reset to zero and the next operand is considered.

After the first address constant is established, it is used as the address constant portion of the relative addresses assigned to subsequent operands.

When the value of the displacement again reaches a value in excess of 4095, another address constant is established. Its value is equal to the current address constant plus the displacement that caused the establishment of the new address constant. This new address constant then becomes current and is used as the address constant for subsequent operands. The displacement is then reset to zero and the next operand is processed. This overall process is repeated until all operands (constants and variables) are processed. Source module arrays are then considered for relative address assignment.

Arrays: Subroutine CORAL-IEKGCR then assigns to each array of the source module that is not in COMMON a relative address that is less than (by the span of the array) the relative address at which the array will reside in the object module. (The concept of span is discussed in Appendix E.) The actual relative address at which an array will reside in the object module is derived from the sum of address constant and displacement that are current at the time the array is considered for relative address assignment. The array span is subtracted from the relative address to facilitate subscript calculations.

Subroutine CORAL-IEKGCR subtracts the span in one of two ways. If the span is less than the current displacement, it subtracts the span from that displacement, and assigns the result as the displacement portion of the relative address for the array. In this case, the address constant assigned to the array is the current address constant. If the span is greater than the current displacement, the CORAL-IEKGCR subroutine subtracts the span from the sum of the current address constant and displacement. The result of this operation is a new address constant, which does not become the current address constant. Subroutine CORAL-IEKGCR assigns the new address constant and a displacement of zero to the array. It then adds the

total size of the array to the location counter, obtains the next array, and tests the value of the displacement. If the value of the displacement does not exceed 4095, the CORAL-IEKGCR subroutine does not take any additional action before it processes the next array. If the displacement value exceeds 4095, the CORAL-IEKGCR subroutine establishes a new address constant, resets the displacement value and processes the next array. After all arrays have relative addresses, subroutine CORAL-IEKGCR calls subroutine EQVAR-IEKGEV to assign address to equivalence variables and arrays that are not in common.

Equivalence Variables and Arrays Not in COMMON: In assigning relative addresses to equivalence variables and arrays, subroutine EQVAR-IEKGEV attempts to minimize the number of required address constants by using, if possible, previously established address constants as the base addresses for equivalence elements. Subroutine EQVAR-IEKGEV processes equivalence information on a group-by-group basis, and assigns a relative address, in turn, to each element of the group. Prior to processing, subroutine EQVAR-IEKGEV determines the base value for the group. The base value is the relative address of the head[1] of the group. The base value equals the sum of the current address constant and displacement (location counter value). After the EQVAR-IEKGEV subroutine has determined the base value, it obtains the first (or next) element of the group and computes its relative address. The relative address for an element equals the sum of the base value for the group and the displacement of the element. The displacement for an element is the number of bytes that the element is displaced from the head of the group (see "COMMON and EQUIVALENCE Processing"). The EQVAR-IEKGEV subroutine then compares the computed relative address to the previously established address constants. If an address constant is such that the difference between the computed relative address and the address constant is less than 4095, the EQVAR-IEKGEV subroutine assigns that address constant to the equivalence element under consideration. The displacement assigned in this case is the difference between the computed relative address of the element and the address constant. Subroutine EQVAR-IEKGEV then processes the next element of the group.

---

[1]The head of an equivalence group is the variable in the group from which all other variables or arrays in the group can be addressed by a positive displacement.

If the desired address constant does not exist, subroutine EQVAR-IEKGEV establishes a new address constant and assigns it to the element. The value of the new address constant is the relative address of the element. The EQVAR-IEKGEV subroutine then assigns the element a displacement of zero, and processes the next element of the group. When all elements of the group are processed, subroutine EQVAR-IEKGEV computes the base value for the next group, if any. This base value is equal to the base value of the group just processed plus the size of that group. The next group is then processed.

COMMON Variables and Arrays: Subroutine EQVAR-IEKGEV considers each COMMON block of the source module, in turn, for relative address assignment. For each COMMON block, subroutine EQVAR-IEKGEV assigns relative addresses to (1) the variables and arrays of that block, and (2) the variables and arrays equivalenced into that COMMON block. (The processing of variables and arrays equivalenced into COMMON is described in a later paragraph.)

Because COMMON blocks are considered separate control sections, the EQVAR-IEKGEV subroutine assigns each COMMON block of the source module a relocatable origin of zero. It achieves the origin of zero by assigning to the first element of a COMMON block a relative address consisting of an address constant and a displacement whose sum is zero. For example, both the address constant and the displacement for the first element in a block can be zero. Also, the address constant can be -16 and the displacement +16. Note that the address constant in the latter case is negative. Negative address constants are permitted, and may be a by-product of the assignment of addresses to COMMON variables and arrays. They evolve from the manner in which the relative addresses are assigned to arrays. A relative address assigned to an array is equal to its actual relative address minus the span of that array. The actual relative address of each array in a common block is equal to the displacement computed for it during COMMON and EQUIVALENCE processing. From the displacement of each array in the COMMON block under consideration, subroutine EQVAR-IEKGEV subtracts the span of that array. The result then replaces the previously computed displacement for the array. If the result of one or more of these computations yields a negative value, the EQVAR-IEKGEV subroutine uses the most negative as the initial address constant for the COMMON block. It then assigns each element (variable or array) in the COMMON block a displacement. This displacement is equal to the absolute

value of the address constant plus the relative address of the element.

If the computations that subtract spans from displacements do not yield a negative value, subroutine EQVAR-IEKGEV establishes an address constant with a value of zero as the initial address constant for the COMMON block. It then assigns each element in the block a relative address consisting of the address constant (with zero value) and a displacement equal to the displacement of the element.

If at any time the displacement to be assigned to an element exceeds 4095, the EQVAR-IEKGEV subroutine establishes a new address constant. This address constant then becomes the current address constant and is saved for inclusion in subsequently assigned addresses. After the new address constant is established, the relative address assigned to each subsequent element consists of the current address constant and a displacement equal to the displacement of that element minus the value of the current address constant. After the entire common block is processed, variables and arrays that are equivalenced into that common block are assigned relative addresses.

Variables and Arrays Equivalenced into Common: Subroutine EQVAR-IEKGEV processes variables and arrays that are equivalenced into common in much the same manner as those that are equivalenced, but not into common. However, in this case, the base value for the group is zero. Only those address constants established for the common block into which the variables and arrays are equivalenced are acceptable as address constants for those variables and arrays.

Adcon and Base Variable Assignment: As CORAL establishes a new address constant and enters it into the adcon table, it also places an entry in the information table. This special entry, called an "adcon variable," points to the new address constant. All operands that have been assigned relative addresses will have pointers to the adcon variable for their address constant. The adcon variables generated for operands are assigned coordinates, via the MCOORD vector and the MVD table. Coordinates 81 through 128 are reserved for base variables; however, some base variables may be assigned coordinates less than 81 if less than 80 coordinates are assigned during the gathering of variable and constant usage information (see PHAZ15, "Gathering Constant/ Variable Usage Information"). Having been assigned coordinates, the adcon variables are now called base variables. Only those operands receiving coordinate assignments are available for full register assignment during phase 20.

Rechaining Data Text

During the assignment of relative addresses to variables, subroutine IEKGCZ rechains the data text entries. Their previous chaining (set by phase 10) was according to their sequence in the source program. The IEKGCZ subroutine now chains the data text entries according to the sequence of relative addresses it assigns to variables. Thus, data text entries are now chained in the same relative sequence in which the variables will appear in the object module. This sequence simplifies the generation of text card images by phase CORAL.

DEFINE FILE Statement Processing

If the source module contains DEFINE FILE statements, subroutine DFILE-IEKTDF converts phase 10 define file text to object-time parameters. These parameters provide the Library routine IHCFDIOSE with the information required to implement direct access READ, WRITE, and FIND statements.

A parameter entry is made for each unit specified in a DEFINE FILE statement. This entry contains the unit number, the relative address of the number of records, a character ('L', 'E', or 'U') indicating the type of formatting to be used, the relative address of the maximum record size, an indicator for the size (four bytes or two bytes) of the associated variable, and the relative address of the associated variable.

Subroutine DFILE-IEKTDF places the parameter entries along with their relative addresses into TXT records. It also places the relative address of the first define file entry into the communication table for later use by phase 25.

NAMELIST Statement Processing

If the source module contains READ/WRITE statements using NAMELIST statements, subroutine NLIST-IEKTNL converts phase 10 namelist text to object-time namelist dictionaries. The object-time namelist dictionaries provide the Library routine IHCFCOMH with the information required to

implement READ/WRITE statements using namelists (see Appendix A, "Namelist Dictionaries"). The dictionary developed for each list in a NAMELIST statement contains the following:

- An entry for the namelist name.
- Entries for the variables and arrays associated with the namelist name.
- An end mark of zeros terminating the list.

Each entry for a variable contains the name, mode (e.g., integer*2 or real*4), and relative address of the variable. Both the address and the mode are obtained from the dictionary entry for the variable.

Each entry for an array contains the name of the array, the mode of its elements, the relative address of its first element, and the information needed to locate a particular element of the array. Subroutine NLIST-IEKTNL obtains the foregoing information from the information table.

The NLIST-IEKTNL subroutine places the entries of the namelist dictionary along with their relative addresses into TXT records. It also places the relative address of the beginning of the namelist dictionary into the address constant for the namelist name.

Initial Value Assignment

CORAL assigns the initial values specified for variables and arrays in phase 15 data text in the following manner:

1. The relative address of the variable or array to be assigned an initial value(s) is obtained and placed into the address field of a TXT record.
2. Each constant (one per variable) that has been specified as an initial value for the variable or array is then obtained and entered into a TXT record. (A number of TXT records may be required if an array is being processed.)

Such action effectively assigns the initial value, because the relative address of the initial value has been set to equal the relative address of its associated variable or array element.

Reserving Space in the Adcon Table

After relative address assignment is completed, subroutine CORAL-IEKGCR calls the IEKTLOAD subroutine (via IEKGCZ) to place an adcon in the object module for special references. Subroutine CORAL-IEKGCR scans the operands of the information table to detect any of these references: call-by-name variables, names of library routines, namelist names, and external references. The byte-A and byte-B usage fields of each information table entry informs subroutine CORAL-IEKGCR whether or not a particular reference belongs to one of these categories. For each special reference that the CORAL-IEKGCR subroutine detects, subroutine IEKGCZ calls subroutine IEKTLOAD to place the needed address constants in the reserved spaces of the object module.

Creating Relocation Dictionary Entries

The relocation dictionary is composed of entries for the address constants of the object module. One relocation dictionary entry (an RLD record) is constructed by subroutine CORAL-IEKGCR for each address it encounters. If the address constant is for an external symbol, the RLD record identifies the address constant by indicating:

- The control section to which the address constant belongs.
- The location of the address constant within the control section.
- The symbol in the external symbol dictionary whose value is to be used in the computation of the address constant.

If the address constant is for a local symbol (i.e., a symbol that is located in the same control section as the address constant), the RLD record identifies the address constant by indicating the control section to which the address constant belongs and its location within that section.

For a more detailed discussion of the use and format of an RLD record, refer to the publication IBM System/360 Operating System: Linkage Editor, Program Logic Manual, Form Y28-6610.

Creating External Symbol Dictionary Entries

The external symbol dictionary contains entries for external symbols that are defined or referred to within the module. An external symbol is one that is defined in one module and referred to in another. One external symbol dictionary entry (an ESD record) is constructed by subroutine IEKGCZ for each external symbol it encounters. The entry identifies the symbol by indicating its type and location within the module. The ESD records constructed by subroutine IEKGCZ are:

- ESD-0 -- This is a section definition record and an entry point definition record for the source module being compiled.
- ESD-2 -- This record is generated for an external subprogram name.
- ESD-5 -- This record is a section definition record for a common block (either named or blank).

For a more complete discussion of the use and the format of these records, refer to the publication IBM System/360 Operating System: Linkage Editor, Program Logic Manual.

PHASE 20

The primary function of phase 20 is to produce a more efficient object module (perform optimization). However, even if the applications programmer has specified no optimization, phase 20 assigns registers for use during execution of the object module.

For a given compilation, the applications programmer may specify OPT=0 (no optimization), or either of the following levels of optimization: OPT=1 or OPT=2. Thus, the functions performed by phase 20 depend on the optimization specified for the compilation.

- If no optimization (OPT=0) has been specified, phase 20 assigns to intermediate text entry operands the registers they will require during object module execution (this is called basic register assignment). As part of this function, phase 20 also provides information about the operands needed by phase 25 to generate machine instructions. Both functions are implemented in a single, block-by-block, top-to-bottom (i.e., according to the order of the statement number chain), pass over the phase 15 text output. The end result of this processing is that the register and status fields of the phase 15 text entries are filled in with the information required by phase 25 to convert the text entries to machine language form (see Appendix B, "Phase 20 Intermediate Text Modifications"). Basic register assignment does not take full advantage of the available general and floating-point registers, and it does not specify the generation of machine instructions that keep operand values in registers (wherever possible) for use in subsequent operations involving them.
- If the OPT=1 level of optimization has been specified, two processes are carried out:
  1. The first process, called full register assignment, performs the same two functions as basic register assignment. However, full register assignment takes greater advantage of available registers and provides information that enables machine instructions to be generated that keep operand values in registers for subsequent operations. An attempt is also made to keep the most frequently used operands in registers throughout the execution of the object module. Full register assignment requires a number of passes over the phase 15 text. The basic unit operated upon is the text block (see Phase 15, "Text Blocking"). The end result of full register assignment, like that of basic register assignment, is that the register and status fields of the phase 15 text entries are filled in with the information required by phase 25.
  2. The second process, called branch optimization, generates RX-format branch instructions in place of RR-format branch instructions wherever possible. The use of RX-format branches eliminates the need for an instruction to load the branch address into a general register. However, branch optimization first requires that the sizes of all text blocks in the module be determined so that the branch address can be found.
- If the OPT=2 level of optimization has been specified, optimization is performed on a "loop-by-loop" basis. Therefore, before processing can be initiated, phase 20 must determine the

structure of the source module in terms of the loops within it and the relationships (nesting) among the loops. Then phase 20 determines the order in which loops are processed, beginning with the innermost (most frequently executed) loop and proceeding outward. The second level of optimization involves three general procedures:

1. The first, called text optimization, eliminates unnecessary text entries from the loop being processed. For example, redundant text entries are removed and, wherever possible, text entries are moved to outer loops, where they will be executed less often.

2. The second procedure is full register assignment, which is essentially the same as in the first level of optimization, but is more effective, because it is done on a loop-by-loop basis.

3. The final procedure is branching optimization, which is the same as in the OPT=1 path.

CONTROL FLOW

In phase 20, control flow may take one of three possible paths, depending on the level of optimization chosen (see Chart 10). Phase 20 consists of a control routine (LPSEL-IEKPLS) and six routine groups. (Table 12 is a directory of the subroutines used by these six groups. In addition, Table 13 contains the list of utility routines called by the subroutines in the various groups.) The control routine controls execution of the phase. All paths begin and end with the control routine. The first group of routines performs basic register assignment. This group is executed only in the control path for non-optimized processing. The second group performs full register assignment. Control passes through this group in the paths for both levels of optimization. The third group of routines performs branch optimization and is also used in the paths for both levels of optimization. The fourth group determines the structure of the source module and is used only in the path for OPT=2 optimization. The fifth group performs loop selection and again is only executed in OPT=2 optimization. The final group performs text optimization and is used only in OPT=2 optimization.

The control routine governs the sequence of processing through phase 20. The processing sequence to be followed is determined from the optimization level specified by the FORTRAN programmer. If no optimization is specified, the basic register assignment routines are brought into play. The unit of processing in this path is the text block. When all blocks are processed, the control routine passes control to the FSD, which calls phase 25.

When OPT=1 optimization is specified, the control routine passes the entire module to the full register assignment routines and then to the routine that computes the size of each text block and sets up the displacements required for branching optimization. Control is then passed to the FSD.

When the control path for OPT=2 optimization is selected, the unit of processing is a loop, rather than a block. In this case, the control routines initially pass control to the routines of phase 20 that determine the structure of the module. When the structure is determined, control is passed to the loop selection routines, to select the first (innermost) loop to be processed. The control routines then pass control to the text-optimization routines to process the loop. When text optimization for a loop is completed, the control routine marks each block in the loop as completed. This action is taken to ensure that the blocks are not reprocessed when a subsequent (outer) loop is processed. The control routine again passes control to the loop selection routines to select the next loop for text optimization. This process is repeated until text optimization has processed each loop in the module. (The entire module is the last loop.)

After text optimization has processed the entire module, the control routine removes the block-completed marks and control is passed to the loop selection routines to reselect the first loop. Control is then passed to the full register assignment routines. When full register assignment for the loop is complete, the control routine marks each block in the loop as completed and passes control to the loop selection routines to select the next loop. This process is repeated for each loop in the module. (The entire module is the last loop.) When all loops are processed, the control routine passes control to the routine that computes the size of each text block and sets up the displacements required for branching optimization. Control is then passed to the FSD.

REGISTER ASSIGNMENT

Two types of register assignment can be performed by phase 20: basic and full. Before describing either type, the concept of status, which is integrally connected with both types of assignment, is discussed.

Each text entry has associated operand and base address status information that is set up by phase 20 in the status field of that text entry (see Appendix B, "Phase 20 Intermediate Text Modification"). The status information for an operand or base address indicates such things as whether ornot it is in a register and whether or not it is to be retained in a register for subsequent use; this information indicates to phase 25 the machine instructions that must be generated for text entries.

The relationship of status to phase 25 processing is illustrated in the following example. Consider a phase 15 text entry of the form A = B + C. To evaluate the text entry, the operands B and C must be added and then stored into A. However, a number of machine instruction sequences could be used to evaluate the expression. If operand B is in a register, the result can be achieved by performing an RX-format add of C to the register containing B, provided that the base address of C is in a register. (If the base address of C is not in a register, it must be loaded before the add takes place.) The result can then be stored into A, again, provided that the base address of A is in a register.

If both B and C are in registers, the result can be evaluated by executing an RR-format add instruction. The result can then be stored into A. Thus, for phase 25 to generate code for the text entry, it must have the status of operands and base addresses of the text entry.

The following facts about status should be kept in mind throughout the discussions of basic and full register assignment:

1. Phase 20 indicates to phase 25 when it is to generate code that loads operands and base addresses into registers, whether or not it is to generate code that retains operands and base addresses in registers, and whether or not operand 1 is to be stored.
2. Phase 20 notes the operands and base addresses that are retained in registers and are available for subsequent use.

Basic Register Assignment -- OPT=0

Basic register assignment involves two functions: assigning registers to the operands of the phase 15 text entries and indicating the machine instructions to be generated for the text entries. In performing these functions, basic register assignment does not use all of the available registers, and it restricts the assignment of those that it does use to special types of items (i.e., operands and base addresses). The registers assigned during basic register assignment and the item(s) to which each is assigned are outlined in Table 3.

Basic register assignment essentially treats System/360 as though it had a single branch register, a single base register, and a single accumulator. Thus, operands that are branch addresses are assigned the branch register, base addresses are assigned the base register, and arithmetic operations are performed using a single accumulator. (The accumulator used depends upon the mode of the operands to be operated upon.)

The fact that basic register assignment uses a single accumulator and a single base register is the key to understanding how text entries having an arithmetic operator are processed. To evaluate the arithmetic interaction of two operands using a single accumulator, one of the operands must be in the accumulator. The specified operation can then be performed by using an RX-format instruction. The result of the operation is formed in the accumulator and is available for subsequent use. Note that in operations of this type, neither of the interacting operands remains in a register.

Table 3. Base and Operand Register Assignment (OPT=0)

Register	Use
General Purpose	
0	Integer or logical operand
1	Integer or logical operand
2	Not assigned
3	Not assigned
4	Integer mult. for subscripting
5	1. Branch register 2. Increment and comparand (BT and BF) 3. Operand 3 (I*2 divide) 4. Integer mult. for subscripting
6	1. Operand representing an index value 2. Secondary spill base for data 3. Spill base for branching (BT and BF)
7	Primary spill base for data
8	Logical result of compare operations
9	Not assigned
10	Not assigned
11	Not assigned
12	Secondary reserved base register
13	Primary reserved base register
14	1. Number of elements (computed GO TO) 2. Spill base for branching (computed GO TO) 3. Branch register (computed GO TO)
15	Index (computed GO TO)
Floating-Point	
0	1. Real operand 2. Real part of complex function result
2	imaginery part of complex function result

Applying this concept to the processing of text entries that are arithmetic in nature, consider that a phase 15 text entry representing the expression A = B + C is the first of the source module. For this text entry to be evaluated using a single accumulator and base register, basic register assignment must tell phase 25 to generate machine code that:

- Loads the base address of B into the base register.
- Loads B into the accumulator.
- Loads the base address of C into the base register. (This instruction is not necessary if C is assigned the same base address as B.)
- Adds C to the accumulator (RX-format add).
- Loads the base address of A into the base register (if necessary).
- Stores the accumulated result in A.

If this coding sequence were executed, two items would remain in registers: the last base address loaded and the accumulated result. These items are available for subsequent use.

Now consider that a text entry of the form D = A + F immediately follows the above text entry. In this case, A, which corresponds to the result operand of the previous text entry, is in the accumulator. Thus, for this text entry, basic register assignment specifies code that:

- Loads the base address of F into the base register. (If the base address of F corresponds to the last loaded base address, this instruction is not necessary.)

- Adds F to the accumulator (RX-format add).

- Loads the base address of D into the base register (if necessary).

- Stores the accumulated result in D.

The foregoing coding sequences are the basic ones specified by basic register assignment for arithmetic operations. The first is specified for text entries in which neither operand 2 nor operand 3 (see Table 3) corresponds to the result operand (operand 1) of the preceding text entry. The second is specified for text entries in which either operand 2 or operand 3 corresponds to the result operand. If operand 3 corresponds to the result operand, the two operands exchange roles, except for division. In the case of division, operand 3 is always in main storage.

If both operands 2 and 3 correspond to the result operand of the previous text entry, an RR-format operation is specified to evaluate the interactions of the operands.

In the actual process of basic register assignment, a single pass is made over the phase 15 text output. The basic unit operated upon is the text block. As the processing of each block is completed, the next block is processed. When all blocks are processed, control is returned to the FSD.

Text blocks are processed in a top-to-bottom manner, beginning with the first text entry in the block. When all text entries in a block are processed, the next text block is processed similarly.

For any text entry, the machine code to be generated is first specified by setting up the status field of the text entry. Registers are then assigned to the operands and base addresses by filling in the register fields of the text entry.

Status Setting: Subroutine SSTAT-IEKRSS sets the operand and base address status information for a text entry in the following order: operand 2, operand 2 base address, operand 3, operand 3 base address, operand 1, and operand 1 base address.

To set the status of operand 2, subroutine SSTAT-IEKRSS determines the relationship of that operand to the result operand (operand 1) of the previous text entry. If operand 2 is the same as the result operand, the SSTAT-IEKRSS subroutine sets the status of operand 2 to indicate that it is in a register and, therefore, need not be loaded; otherwise, it sets the status to indicate that it is in main storage. Subroutine SSTAT-IEKRSS uses a similar procedure to set the status of operand 3.

To set the status of the base address of operand 2, subroutine SSTAT-IEKRSS determines the relationship of that base address to the current base address (see note). Ifthey correspond, the SSTAT-IEKRSS subroutine sets the status of the base address of operand 2 to indicate that it is in a register and, therefore, need not be loaded; otherwise, it sets the status to indicate that it is in main storage.

Subroutine SSTAT-IEKRSS sets the statuses of the base addresses of operands 3 and 1 in a similar manner.

Note: The current base address is the last base address loaded for the purpose of referring to an operand. This base address remains current until a subsequent operand that has a different base address is encountered. When this occurs, the base address of the subsequent operand must be loaded. That base address then becomes the current base address, etc.

The SSTAT-IEKRSS subroutine sets status of operand 1 to indicate whether or not the result of the interaction of operands 2 and 3 is to be stored into operand 1. If operand 1 is either an actual operand (a variable defined by the programmer) or a temporary that is not used in the subsequent text entry, it sets the status of operand 1 to indicate that the store operation is to be performed; otherwise, it sets the status to indicate that a store into operand 1 is unnecessary.

Register Assignment: After the status field of the text entry is completed, subroutine SPLRA-IEKRSL assigns registers to the operands of the text entry and their associated base addresses in the same order in which statuses were set for them.

The assignment of registers depends upon the statuses of the operands of the text entry. To assign a register to operand 2,

subroutine SPLRA-IEKRSL examines the status of that operand, and, if necessary, of operand 3. If the status of operand 2 indicates that it is in a register or if the statuses of operands 2 and 3 indicate that neither is a register, subroutine SPLRA-IEKRSL assigns operand 2 to a register. It selects the register according to the type of operand (see Table 3), and places the number of that register into the R2 field of the text entry.

To assign a register to the base address of operand 2, subroutine SPLRA-IEKRSL determines the status of operand 2. If the status of that operand indicates that it is not in a register, it assigns a register to the base address of operand 2. The appropriate register is selected as shown in Table 3, and the register number is placed into the B2 field of the text entry. If the status of operand 2 indicates that it is in a register, subroutine SPLRA-IEKRSL does not assign a register to the base address of operand 2. The SPLRA-IEKRSL subroutine uses a similar procedure in assigning a register to the base address of operand 3.

If the status of operand 3 indicates that it is in a register, subroutine SPLRA-IEKRSL assigns the appropriate register (see Table 3) to that operand, and enters the number of that register into the R3 field.

Operand 1 is always assigned a register. Subroutine SPLRA-IEKRSL selects the register according to the type of operand 1 (see Table 3), and places the number of that register into the R1 field.

The base address of operand 1 is assigned a register only if the status of operand 1 indicates the result is to be stored into operand 1. If such is the case, subroutine SPLRA-IEKRSL selects the appropriate register, and records the number of that register in the B1 field. If the status of operand 1 indicates that the result is not to be stored into operand 1, subroutine SPLRA-IEKRSL does not assign a register to the base address of operand 1.

When all the operands of the text entry and their associated base addresses are assigned registers, the next text entry is obtained, and the status setting and register assignment processes are repeated. After all text entries in the block are processed, control is returned to IEKRSS, which then makes the next block available to the basic register assignment routines. When the processing of all blocks is completed, control is passed to IEKPLS, and then to the FSD.

Full Register Assignment -- OPT=1 (Chart 14)

During full register assignment (also refer to "Full Register Assignment -- OPT=2"), as during basic register assignment, registers are assigned to the text entry operands and their associated base addresses, and the machine code to be generated for the text entries is specified. To improve object module efficiency, these functions are performed in a manner that reduces the number of instructions required to load base addresses and operands. This process reduces the number of required load instructions by taking greater advantage of all available registers, by assigning the registers as needed to both base addresses and operands, by keeping as many operands and base addresses as possible in registers and available for subsequent use, and by keeping the most active base addresses and operands in registers where they are available for use throughout execution of the entire object module.

During full register assignment, registers are assigned at two levels: "locally" and "globally." Local assignment is performed on a block-by-block basis. Global assignment is performed on the basis of the entire module (if intermediate optimization has been specified).

For local assignment, an attempt is made to keep operands whose values are defined within a block in registers and available for use throughout execution of that block. This is done by assigning an available register to an operand at the point at which its value is defined. (The value of an operand is defined when that operand appears in the operand 1 position of a text entry.) The same register is assigned to subsequent uses (i.e., operand 2 or operand 3 appearances) of that operand within the block, thereby ensuring that the value of the operand will be in the assigned register and available for use. However, if more than one subsequent use of the defined operand occurs in the block, additional steps must be taken to ensure that the value of that operand is not destroyed between uses. Thus, when the text entries in which the defined operand is used are processed, the code specified for them must not destroy the contents of

the register containing the defined operand.

Because all available registers are used during full register assignment, a number of operands whose values are defined within the block can be retained in registers at the same time.

Applying the above concept to an example, consider the following sequence of phase 15 text entries;

```
A = X + Y
C = A + Z
F = A + C
```

A register is assigned to A at the point at which its value is defined, namely in the text entry A = X + Y. The same register is assigned to the subsequent uses of A. The value of A will be accumulated in the assigned register and can be used in the subsequent text entry C = A + Z. However, because A is also used in the text entry F = A + C, the contents of the register containing A cannot be destroyed by the code generated for the text entry C = A + Z. Thus, when the text entry C = A + Z is processed, instructions are specified for that text entry that use the register containing A, but that do not destroy the contents of that register.

In the example, C is also defined and subsequently used. To that defined operand and its subsequent uses, a register is assigned. The assigned register is different from that assigned to A. The value of C will be accumulated in the assigned register and can be used in the next text entry. The text entry F = A + C can then be evaluated without the need of any load operand instructions, because both the interacting operands (A and C) are in registers.

This type of processing typifies that performed during local assignment for each block. When all blocks are processed, global assignment for the source module is carried out.

Global assignment increases the efficiency of the object module as a whole by assigning registers to the most active operands and base addresses. The activities of all operands and base addresses are computed during local assignment prior to global assignment. The first register available for global assignment is assigned to the most active operand or base address; the next available register is assigned to the next most active operand or base address; etc. As each such operand or base address is processed, a text entry, the function of which is to load the operand or base address into the assigned register, is generated and placed into the entry block(s) of the module. When the supply of operands and base addresses, or the supply of available registers, is exhausted, the process is terminated.

All global assignments are recorded for use in a subsequent text scan, which incorporates global assignments into the text entries, and completes the processing of operands that have neither been locally nor globally assigned to registers (e.g., an infrequently used operand that is used in a block but not defined in that block).

The full register assignment process is divided into five areas of operation: control (subroutine REGAS-IEKRRG), table building (subroutine FWDPAS-IEKRFP), local assignment (subroutine BKPAS-IEKRBP), global assignment (subroutine GLOBAS-IEKRGB), and text updating (subroutine STXTR-IEKRSX). The control routine of phase 20 (LPSEL-IEKPLS) passes control to subroutine REGAS-IEKRRG that directs the flow of control among the other full register assignment routines.

The actual assignment of registers is implemented through the use of tables built by the table-building routine, with assistance from the control routine. Tables are built using the set of coordinate numbers and associated dictionary pointers created by phase 15 (the MCOORD vector and MVD) for indexing. The table-building routine constructs two sets of parallel tables. One set, used by the local assignment routine, contains information about a text block; the second set, used by the global assignment routines, contains information about the entire module. (The local assignment and global assignment tables are detailed in Appendix A, "Register Assignment Tables.")

The flow of control through the full register assignment routines is, as follows:

1. The control routine (REGAS-IEKRRG) makes a pass over the MVD table and the dictionary entries for the variables and constants in the loop passed to it, and constructs the eminence table (EMIN) for the module,

which indicates the availability of the variables for global assignment. Then the REGAS-IEKRRG subroutine calls the table building routine to process the blocks in the loop (the complete module for OPT=1).

2. The table-building routine (FWDPAS-IEKRFP) builds the required set of local assignment tables and adds information to the global assignment tables under construction. Subroutine FWDPAS-IEKRFP selects the first block of the loop and builds the tables for that block. It then passes control to the local assignment routine to process the block and the tables (see Chart 15).

3. The local assignment routine (BKPAS-IEKRBP) uses the tables supplied for the block to perform local register assignment, and returns control to subroutine FWDPAS-IEKRFP when its processing is completed (see Chart 16).

4. The FWDPAS-IEKRFP subroutine selects the next block of the loop and again builds tables. This process continues until all blocks of the loop have been processed. Control is then returned to the REGAS-IEKRRG subroutine.

5. Subroutine REGAS-IEKRRG passes control to the global assignment routine GLOBAS-IEKRGB, which performs global assignment for the module (see Chart 17).

6. When global assignment is complete, the control routine calls the text updating routine, STXTR-IEKRSX, to complete register assignment by entering the results of global assignment into the text entries for the module. Control is then returned to the LPSEL-IEKPLS subroutine.

Table Building for Register Assignment (Chart 15): The table-building routine, FWDPAS-IEKRFP, performs a forward scan of the intermediate text entries for the block under consideration and enters information about each text entry into the local and global tables (see Appendix A, "Register Assignment Tables"). The local assignment tables can accommodate information for 100 text entries. If, however, a block contains more than 100 text entries, the table-building routine builds the local tables for the first 100 text entries and passes this set of tables to the local assignment routine. The local assignment routine processes the text entries represented in the set of local tables. The table-building routine then creates the local tables for the next 100 text entries in the block and passes them to the local assignment routine. When the table-building routine encounters the last text entry for the block, it passes control to the local assignment routine, although there may be fewer than 100 entries in the local tables.

The global tables contain information relating to variables and constants referred to within the module, rather than to text entries. The global tables can accommodate information for 126 variables and constants in a given module. Variables and constants in excess of this number within the module are not processed by the global assignment routine.

Local Assignment (Chart 16): Local assignment is implemented via a backward pass over the text items for the block (or portion of a block) under consideration. The text items are referred to by using the local assignment tables, which supply pointers to the text items.

The local assignment routine, BKPAS-IEKRBP, examines each operand in the text for a block and determines (from the local assignment tables) whether or not the operand is eligible for local assignment. To be eligible, an operand must be defined and used (in that order) within a block. Because local assignment is performed via a backward pass over the text, an eligible operand will be encountered when it is used (i.e., in the operand 2 or 3 position) before it is defined.

When an operand of a text entry is examined, the local assignment routine (BKPAS-IEKRBP) consults the local assignment tables to determine that operand's eligibility. If the operand is eligible, subroutine BKPAS-IEKRBP assigns a register to it. The register assigned is determined by consulting the register usage table for local assignment (TRUSE). TRUSE is a work table that contains an entry for every register that may be used by the local assignment routine. A zero entry for a particular register indicates that the register is available for local assignment. A nonzero entry indicates that the register is unavailable and identifies the variable

to which the register is assigned. The register usage table is modified each time a register is assigned or freed. The first time a register is assigned, a corresponding entry in the register usage table for global assignment (RUSE) is set. This entry implies that the register is unavailable for global assignment.

Subroutine BKPAS-IEKRBP records the register assigned to the used operand in the local assignment tables and in the text item containing the used operand. It sets the status of the operand in the text entry to indicate that it is in a register. If subsequent uses of the operand are encountered prior to the definition of the operand, the BKPAS-IEKRBP subroutine uses the register assigned to the first use, and records its identity in the text item. It then sets the status bits for the operand to indicate that it is in a register and is to be retained in that register.

When a definition of the operand is encountered, subroutine BKPAS-IEKRBP enters the register assigned to the operand into the text item and sets the status for the operand to indicate its residence in a register. Once the register is assigned to the operand at its definition point, the BKPAS-IEKRBP subroutine frees the register by setting the entry in the register usage table to zero, making the register available for assignment to another operand.

If the block being processed contains a CALL statement or a reference to a function subprogram, common variables, arguments, and real operands cannot be assigned to registers across that reference. The local assignment routine assumes that:

1. All mathematical functions return the result in general register 0 or floating-point register 0, according to the mode of the function.

2. The imaginary portion of a complex result is returned in floating-point register 2.

If no register is available for assignment to an eligible operand, an overflow condition exists. In this case, subroutine BKPAS-IEKRBP must free a previously assigned register for assignment to the current operand. It scans the local assignment tables and selects a register. It then modifies the local assignment tables, text entries for the block, and register usage table to negate the previous assignment of the selected register. The required register is now available, and processing continues in the normal fashion.

Global Assignment (Chart 17): The global assignment routine (GLOBAS-IEKRGB), unlike the local assignment routine, does not process any of the text entries for the module. The global assignment routine operates only through the set of global tables. The results of global assignments are entered into the appropriate text entries by the text updating routine.

Before assigning registers, the global assignment routine modifies the global assignment tables to produce a single activity table for all operands and base addresses in the module.

Global assignment is then performed based on the activity of the eligible operands and base addresses.

The GLOBAS-IEKRGB routine determines the eligibility of an operand or base address by consulting the appropriate entry in the global assignment tables. Eligible operands are divided into two categories: floating point and fixed point. The two categories are processed separately, with floating-point quantities processed first.

The register usage table for global assignment (RUSE) is of the same type as described under local assignment (TRUSE). For each category of operands, the GLOBAS-IEKRGB routine selects the eligible operand with the highest total activity and assigns it the first available register of the same mode. It records the assignment in the register usage table and in the global assignment tables. The GLOBAS-IEKRGB routine then selects the eligible operand with the next highest activity and treats it in the same manner. Processing for each group continues until the supply of eligible operands or the supply of available registers is exhausted.

If the module contains any CALL statements or function subprogram references, arguments and real and common variables are ineligible for global assignment. In other words, if a module contains either a reference to a subroutine or to a function subprogram, global assignment is restricted to integer and logical operands that are not in common or in the parameter list.

Text Updating (Charts 18 and 19): The text updating routine (STXTR-IEKRSX) completes full register assignment. It scans each text entry within the series of blocks comprising the module, looking at operands 2, 3, and 1, in that order, within each text entry. As each operand is processed, subroutine STXTR-IEKRSX interrogates the completed global assignment table to determine whether or not a global assignment has been made for the operand.

If it has, subroutine STXTR-IEKRSX enters the register assigned into the text entry and sets the operand status bits to indicate that the operand is in a register and is to be retained in that register.

If both a local and a global assignment have been made for an operand, the global assignment supersedes the local assignment and the STXTR-IEKRSX subroutine records the globally assigned register in the text items pertaining to that operand. It also sets the status bits for such an operand to indicate that it is in a register and is to be retained in that register.

If a register has not been assigned either locally or globally for an operand, subroutine STXTR-IEKRSX determines and records in the text entry the required base register for the base address of that operand. If the base address corresponds to one that has been assigned to a register during global assignment, the STXTR-IEKRSX subroutine assigns the same register as the base register for the operand. If a register has not been assigned to the base address of the operand during global assignment, it assigns a spill register (register 15) as the base register of the operand. Subroutine STXTR-IEKRSX sets the operand's base status bits to indicate whether or not the base address is in a register. (The base address will be in a register if one was assigned to it during global assignment.) It then assigns the operand itself a spill register (general register 0 or 1 or floating-point register 0, depending upon its mode).

As part of its text updating function, subroutine STXTR-IEKRSX allocates temporary storage where needed for temporaries that have not been assigned to a register, keeps track of the allocated temporary storage, and completes the register fields of text entries to ensure compatibility with phase 25. On exit from the text updating routine, all text items in the module are fully formed and ready for processing by phase 25. The text updating routine returns control to subroutine REGAS-IEKRRG upon completion of its functions. The REGAS-IEKRRG subroutine, in turn, returns control to subroutine LPSEL-IEKPLS.

BRANCHING OPTIMIZATION -- OPT=1

This portion of phase 20 optimizes branching within the object module. The optimization is achieved by generating RX-format branch instructions in place of RR-format branch instructions wherever possible.

The use of RX-format branches eliminates the need for an instruction to load the branch address into a general register preceding each branching instruction. Thus, branching optimization decreases the size of the object module by one instruction for each RR-format branch instruction in the object module that can be replaced by an RX-format branch instruction. It also decreases the number of address constants required for branching.

Phase 20 optimizes branching instructions by calculating the size of each text block (number of bytes of object code to be generated for that block) and by determining those blocks that can be branched to via RX-format branch instructions.

Subroutine BLS-IEKSBS calculates the sizes of all text blocks after full register assignment for the module is completed. It then uses the gathered block size information to determine the blocks to which a branch can be made by means of RX-format branch instructions. The BLS-IEKSBS subroutine calculates the number of bytes of object code by:

1. Examining each text item operation code and the status of the operands (i.e., in registers or not).

2. Determining, from a reference table, the number of bytes of code that is to be generated for that text item.

The BLS-IEKSBS subroutine accumulates these values for each block in the module. In addition, it increments the block size count by the appropriate number of bytes for each reference to an in-line routine that it encounters.

Next, subroutine BLS-IEKSBS computes all block sizes and determines those text blocks to which a branch can be made via RX-format branch instructions. Once converted to machine code, a branch can be made to a text block via an RX-format branch instruction if the relative address of the beginning of that block is displaced less than 4096 bytes from an address that is loaded into a reserved register.

The following text discusses reserved registers, the addresses loaded into them, and the processing performed by subroutine BLS-IEKSBS to determine the source module blocks to which a branch can be made via RX-format branch instructions.

Reserved Registers

Reserved registers are allocated to contain the starting address of the adcon table and subsequent 4096-byte blocks of the object module. The criterion used by phase 20 in reserving registers for this purpose is the number of text entries that result from phase 15 processing. (Phase 15counts the number of text entries that result from its processing and passes the information to phase 20.) For small source modules (up to 880 text entries), phase 20 reserves only one register in addition to register 13. For large source modules (more than 1760 text entries), a maximum of four additional registers is reserved. The registers are reserved, as needed, in the following order: register 13, 12, 11, 10, and 9.

Reserved Register Addresses

The addresses placed into the reserved registers as a result of the execution of the initialization instructions (see "Generation of Initialization Instructions" under "FORTRAN System Director") are:

- Register 13 -- address of the save area.
- Register 12 (if reserved) -- address of the save area plus 4096 or address of the first adcon for the program.
- Register 11 (if reserved) -- address of the register 12 plus 4096.
- Register 10 (if reserved) -- address of the register 12 plus 2(4096).
- Register 9 (if reserved) -- address of the register 12 plus 3(4096).

Block Determination and Subsequent Processing

Because the instructions resulting from the compilation are entered into text information immediately after the "B" block labels (see Figure 9), certain text blocks are displaced less than 4096 bytes from an address in a reserved register. A branch can be made to such blocks by RX-format branch instructions that use the address in a reserved register as the base address for the branch.

To determine the blocks to which a branch can be made via RX-format branch instructions, subroutine BLS-IEKSBS computes the displacement (using the block size information) of each block from the address in the appropriate reserved register. The first reserved register address considered is that in register 13. For each block that has a displacement of less than 4096 bytes from that address, subroutine BLS-IEKSBS enters the displacement into the statement number entry for that block. It also places in that statement number entry an indication that a transfer can be made to the block via an RX-format branch instruction, and records the number of the reserved register to be used in that branch instruction.

When subroutine BLS-IEKSBS has processed all blocks displaced less than 4096 bytes from the address in register 13, it processes those that are displaced less than 4096 bytes from the addresses in registers 12, 11, 10, and 9 (if reserved) in a similar manner.

The information placed in the statement number entries is used during code generation, a phase 25 process, to generate RX-format branch instructions.

STRUCTURAL DETERMINATION

To achieve OPT=2 optimization, the structural determination routines of phase 20 (TOPO-IEKPO and BAKT-IEKPB) identify module loops and specify the sequence in which they are to be processed. Loops are identified by analyzing the block connection information gathered by phase 15 and recorded in the forward-connection (RMAJOR) and backward-connection (CMAJOR) tables. The connection information indicates the flow of control within the module and, therefore, reflects which blocks pass control among themselves in a cyclical fashion.

Loops are ordered for processing starting with the innermost, or most often executed, loop and working toward the outermost. The inner-to-outer loop sequence is specifed so that:

- Text entries will not be relocated into loops that have already been processed.[1]
- The full register capabilities of System/360 can first be applied to the most frequently executed (innermost) loop.

Loop identification is a sequential process, which requires that a back dominator be determined for each text block. The back dominator of a text block (block I) is defined as the block nearest to block I through which control must pass before block I receives control for the first time. The back dominators of all text blocks must be determined before loop identification can be continued. After all back dominators have been determined, a chain of back dominators is effectively established for each block. This chain consists of the back dominator of the block, the back dominator of the back dominator of the block, etc.

Figure 7 illustrates the concept of back dominators. Each block in the illustration represents a text block. The blocks are identified by single letter names. The back dominator of each block is identified and recorded above the upper right-hand corner of that block.

When all back dominators are identified, a back target and a depth number for each text block is determined. A block (block I) has a back target (block J) if:

- There exists a path from block I to itself that does not pass through block J.
- Block J is the nearest block in the chain of back dominators of block I that has only one forward connection.

The text blocks constituting a loop are identifiable because they have a common back target, known as the back target of the loop.

The depth number for a block indicates the degree to which that block is nested within loops. For example, if a block is an element of a loop that is contained within a loop with a depth number of one, that block has a depth number of two. All blocks constituting the same loop (i.e., all blocks having a common target) have the same depth number.

The depth numbers computed for the blocks that comprise the various loops are used to determine the sequence in which the loops are to be processed.

Figure 8 illustrates the concepts of back targets and depth numbers. Again each block in the illustration represents a text block, which is identified by a single letter name. In this illustration, the back target of each block is identified and recorded above the upper right-hand corner of that block. The depth number for the block is recorded above the upper left-hand corner of the block. Note that blocks that pass control among themselves in a looping fashion have a common back target and the same depth number. Also note that the blocks of the two inner loops have the same depth numbers, although they have different back targets.

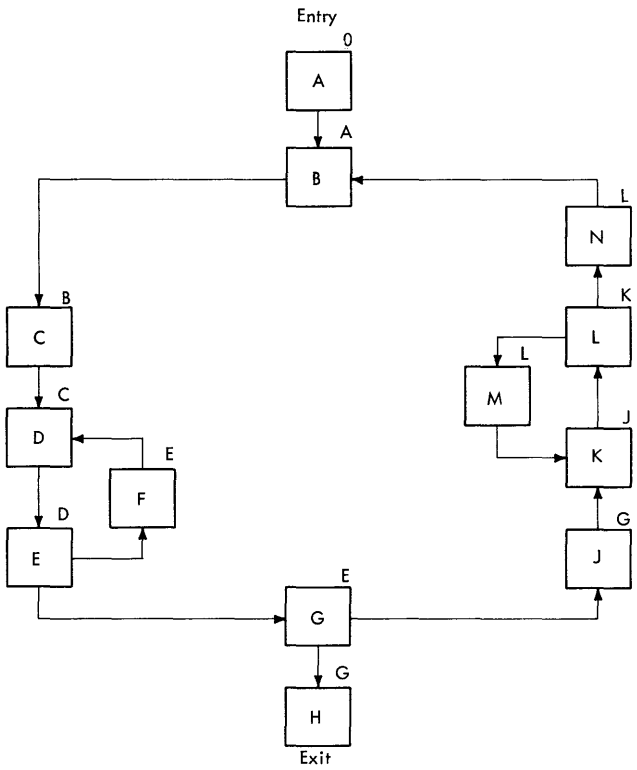


Figure 7. Back Dominators

[1]The text optimization process relocates text entries from within an inner loop to an outer loop. Thus, if an outer loop were processed first, text entries from an inner loop might be relocated to the outer loop, thereby requiring that the outer loop be reprocessed.

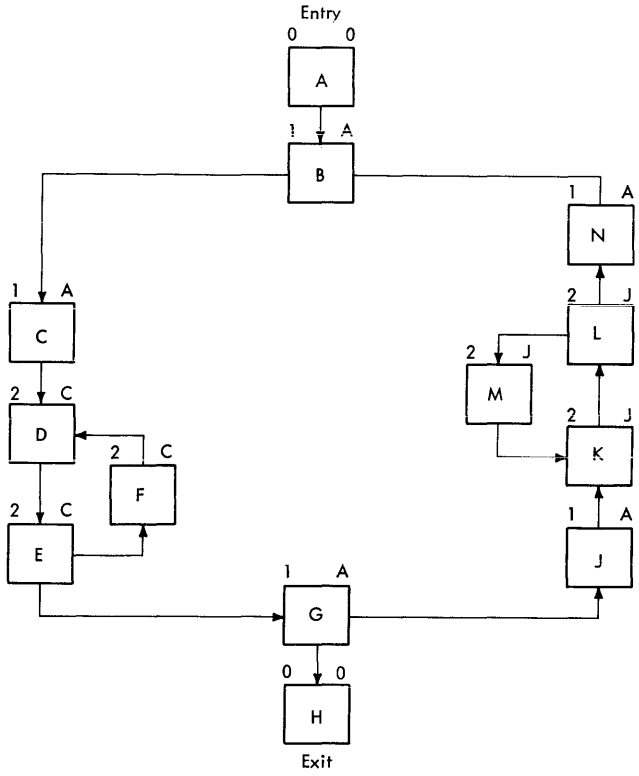


Figure 8. Back Targets and Depth Numbers

When the back target and depth number of each text block has been determined, loops are identified and the sequence in which they are to be processed is specified. The loops are sequenced according to the depth number of their blocks. The loop whose blocks have the highest depth number is specified as the first to be processed, the loop whose blocks have the next highest depth number is specified as the second to be processed, etc. When the processing sequence of all loops has been established, the innermost loop is selected for processing.

The following paragraphs describe the processing performed by the structural determination routines to:

- Determine the back dominator of each text block.
- Determine the back target and depth number of each text block.
- Identify and sequence loops for processing.

Determination of Back Dominators

Subroutine TOPO-IEKPO determines the back dominator of each text block by examining the connection information for that block. The first block processed by subroutine TOPO-IEKPO is the first block (entry block) of the module. Blocks on the first level (i.e., blocks that receive control from the entry block) are processed next. Second-level blocks (i.e., blocks that receive control from first-level blocks) are then processed, etc.

The TOPO-IEKPO subroutine assigns to the entry block a back dominator of zero, because it has no back dominator; it records the zero in the back dominator field of the statement number entry for that block (see Appendix A, "Statement Number/Array Table"). The TOPO-IEKPO subroutine assigns each block on the first level either its actual back dominator or a provisional back dominator. If a first-level block receives control from only one block, that block must be the entry block and is the back dominator for the first-level block. Subroutine TOPO-IEKPO records a pointer to the statement number entry for the entry block in the back dominator field of the statement number entry for the first-level block. If a first-level block receives control from more than one block, subroutine TOPO-IEKPO assigns to it a provisional back dominator, which is the entry block of the module. All blocks on the first level are processed in this manner.

Subroutine TOPO-IEKPO also assigns each block on the second level either its actual back dominator or a provisional back dominator. If a second-level block receives control from only one block, its back dominator is the first-level block from which it receives control. The TOPO-IEKPO subroutine records a pointer to the statement number entry for the first-level block in the back dominator field of the statement number entry for the second-level block. If more than one block passes control to a second-level block, subroutine TOPO-IEKPO assigns to that block a provisional back dominator. The provisional back dominator assigned is a first-level block that passes control to the second-level block under consideration. Processing of this type is performed at each level until the last, or exit, block of the module is processed. Subroutine TOPO-IEKPO then determines the actual back dominators of blocks that were assigned provisional back dominators.

For each block assigned a provisional back dominator, subroutine TOPO-IEKPO makes a backward trace over each path leading to the block (using CMAJOR). The blocks at which two or more of the paths converge are flagged as possible candidates for the back dominator of the block. When all paths

have been treated, the relationship of each possible candidate to the other possible candidates is examined. The TOPO-IEKPO subroutine assigns the candidate at the highest level (i.e., closest to the entry block of the module) as the back dominator of the block under consideration; it records a pointer to the statement number entry for the assigned back dominator in the back dominator field of the statement number entry for the block under consideration. After the back dominators of all text blocks are identified, subroutine BAKT-IEKPB determines the back target and depth number of each text block.

Determination of Back Targets and Depth Numbers

Subroutine BAKT-IEKPB determines the back target of each text block through an analysis of the backward connection information (in CMAJOR) for that block. Block J is the back target of block I if:

1. A path exists from block I to itself, and block J is the nearest block, in the chain of back dominators of block I, not on that path.
2. Block J has only one forward connection.

If a block J exists that satisfies condition 1 but not condition 2, then the back target of block J is also the back target of block I.

If a block J satisfying condition 1 does not exist, then the back target of block I is zero.

When the back target of a block is identified, that block is also assigned a depth number.

Back targets and depth numbers are determined for text blocks in the same sequence as back dominators are determined for them. The first block of the module is the first processed, first-level blocks are considered next, etc.

The BAKT-IEKPB subroutine assigns the first or entry block both a back target and depth number of zero, because it does not have a back target and is not in a loop. It records the depth number (zero) in the loop number field of the statement number entry for the entry block (see Appendix A, "Statement Number/Array Table").

The processing performed by subroutine BAKT-IEKPB for each of the other blocks depends upon whether one or more than one block passes control to that block. If more than one block passes control to the block under consideration, subroutine BAKT-IEKPB makes a backward trace over all paths leading to that block to locate its primary path. The primary path of a block (if one exists) is a path that starts at that block and converges on that block without passing through any block in the chain of back dominators of that block.

If such a path exists, subroutine BAKT-IEKPB obtains and examines the nearest block in the chain of back dominators of the block under consideration. If the obtained block has a single forward connection, subroutine BAKT-IEKPB assigns that block as the back target of the block under consideration. The BAKT-IEKPB subroutine then assigns a depth number to the block. The number is one greater than that of its back target, because the block is in a loop, which must be nested within the loop containing the back target. Subroutine BAKT-IEKPB records the depth number in the loop number field of the statement number entry for the block.

If the obtained block has more than one forward connection, subroutine BAKT-IEKPB assigns its back target as the back target of the block under consideration. The BAKT-IEKPB subroutine then records in the statement number entry for the block a depth number one greater than that of its back target.

If a block that receives control from two or more blocks does not have an associated primary path, that block, if it is in a loop at all, is in the same loop as one of the blocks in its chain of back dominators. To identify the loop containing the block (block I), subroutine BAKT-IEKPB obtains and examines the nearest block to block I in its chain of back dominators that has two or more forward connections. The BAKT-IEKPB subroutine makes a backward trace over all paths leading to the obtained block to determine whether or not block I is an element of such a path. If block I is an element of such a path, it is in the same loop as the obtained block, and subroutine BAKT-IEKPB, therefore, assigns block I the same back target and depth number as the obtained block; it records the depth number in the statement number entry for block I.

If block I is not an element of any path leading to the obtained block, subroutine BAKT-IEKPB obtains the next nearest block to block I in its chain of back dominators that has two or more forward connections and repeats the process. If block I is not an element of any path leading to any block in its chain of back dominators, block I is not in a loop, and the BAKT-IEKPB

subroutine assigns it both a back target and depth number of zero.

A block that receives control from only one block, if it is in a loop at all, is in the same loop as one of the blocks in its chain of back dominators. To identify the loop containing a block (block I) that receives control from only one block, subroutine BAKT-IEKPB obtains and examines the nearest block to block I in its chain of back dominators that receives control from two or more blocks. The BAKT-IEKPB subroutine makes a backward trace over all paths leading to the obtained block to locate its primary path (if any). If the obtained block has a primary path, subroutine BAKT-IEKPB retraces it to determine whether or not block I is an element of the path. If it is, block I is in the same loop as the obtained block, and, BAKT-IEKPB therefore assigns block I the same back target and depth number as the obtained block; BAKT-IEKPB then records the depth number in the statement number entry for block I.

If the obtained block does not have a primary path, or if it does have a primary path, which, however, does not have block I as an element, the BAKT-IEKPB subroutine considers the next nearest block to block I in its chain of back dominators that receives control from two or more blocks. The process is repeated until a primary path containing block I is located (if any such path exists). If block I is not in the primary path of any block in its chain of back dominators, block I is not in a loop and subroutine BAKT-IEKPB assigns it both a back target and depth number of zero.

Identifying and Ordering Loops for Processing

Subroutine BAKT-IEKPB orders blocks for processing on the basis of the determined back target and depth number information. Blocks that have a common back target and the same depth number constitute a loop. The BAKT-IEKPB subroutine flags the loop with the highest depth number (therefore, the most deeply nested loop) as the first loop to be processed. It assigns the blocks constituting that loop a loop number of one, indicating that they form the innermost loop, which is the first to undergo optimization. (Subroutine BAKT-IEKPB records the value 1 in the loop number field of the statement number entry for each block in that loop.) The BAKT-IEKPB subroutine flags the loop with the next highest depth number as the second loop to be processed. It assigns the blocks in that loop a loop number of two, indicating that they form the second (or next outermost) loop to be processed. (A value of 2 is recorded in the loop number field of the statement number entry for each block in that loop.) Subroutine BAKT-IEKPB repeats this procedure until the loop with a depth number of one is processed. It then assigns the highest loop number to the blocks with a depth number of zero, indicating that they do not form a loop.

If at any time, groups of blocks with the same depth number but different back targets are found, each group is in a different loop. Therefore, each such loop is, in turn, processed before blocks having a lesser depth number are considered. Thus, if the blocks of two loops have the same depth number, subroutine BAKT-IEKPB assigns the blocks of the first loop the next loop number. It assigns the blocks of the second loop a loop number one greater than that assigned to the blocks of the first loop.

When loop numbers are assigned to the blocks of all module loops, the sequence in which the loops are to be processed has been specified. Control is passed to the routine that determines the busy-on-exit information and then to the loop selection routine to select the first (innermost) loop to be operated upon. This loop consists of all blocks having a loop number of one.

BUSY-ON-EXIT INFORMATION

Before the module can be processed on a loop-by-loop basis, the variables in each block must be classified as either busy-on-exit from the block or not busy-on-exit from the block. A variable is busy immediately preceding a use of that variable, but is not busy immediately preceding a definition of that variable. Thus, a variable is busy-on-exit from the blocks that are along all paths connecting a use and a prior definition of that variable. This means that in subsequent blocks the variable can be used before it is defined. The busy-on-exit condition for a variable assures that its proper value exists in main storage or in a register along each path in which it is subsequently used.

Information about the regions in which a variable is busy or not busy determines whether or not a definition of that variable can be moved out of a loop. For example, if a variable is busy-on-exit from the back target of a loop, text

optimization (see "Text Optimization") would not attempt to move to the back target a redefinition of that variable, because if moved, the value of the variable, as it is processed along various paths from the back target, might not be the desired value. Conversely, if the variable is not busy-on-exit, the redefinition can be moved without affecting the desired value of the variable. Thus, text optimization respects the redefinitions of variables that are busy-on-exit from the back target of a loop.

The information about regions in which a variable is busy or not busy also determines whether or not load and store operations of a register assigned to the variable are required. For example, in full register assignment (see "Full Register Assignment--OPT=2"), variables that are assigned registers during global assignment and that are busy-on-exit from the back target of the loop must have an initializing load of the register placed into the back target. The load is required because the variable may be used before its value is defined. Conversely, if the globally assigned variable is not busy-on-exit from the back target, an initializing load is unnecessary.

Phase 15 provides phase 20 with not busy-on-entry information for each operand that is assigned a coordinate (an MVD table entry). The not busy-on-entry information is recorded in the MVX field of the statement number text entry for each text block (see Phase 15, "Gathering Constant/Variable Usage Information"). An operand is not busy-on-entry to a block, if in that block that operand is defined but not used or defined before it is used. Phase 20 converts the not busy-on-entry information to busy-on-entry information. An operand is busy-on-entry to a block, if in that block that operand is used but not defined or used before it is defined. Finally, phase 20 converts the busy-on-entry information to busy-on-exit information. The backward-connection information in CMAJOR is used to make the final conversion.

The routine that performs the conversions is BIZX-IEKPZ. This routine determines busy-on-exit information for each constant, variable, and base variable having an associated MVD table entry or coordinate. However, because only constants and base variables are used, they are busy-on-exit throughout the entire module. Therefore, the remainder of this discussion deals with the determination of busy-on-exit information for variables.

Because RETURN statements (exit blocks) and references to subprograms not supplied by IBM constitute implicit uses of variables in common, all common variables and arguments to such subprograms are first marked as busy-on-entry to exit blocks and blocks containing the references. The common variables and arguments are found by examining the information table entries for all variables in the MVD table. The module is then searched for blocks that are exit blocks and that contain references to subprograms not supplied by IBM. The coordinate bit for each previously mentioned variable is set to on in the MVF field of the statement number text entry for each such block, while the same coordinate bit in the MVX field is set to off. This defines the variable to be busy-on-entry to such a block. During this process, a table, consisting of pointers to exit blocks, is built for subsequent use.

After the previously discussed blocks have been appropriately marked for common variables and arguments, subroutine BIZX-IEKPZ, working with the coordinate assigned to a variable, converts the not busy-on-entry information for the variable to a table of pointers to blocks to which the variable is busy-on-entry. (The not busy-on-entry information for the variable is contained in the MVX fields of the statement number text entries for the various text blocks.) At the same time, the variable's coordinate bit in each MVX field is set to off. The busy-on-entry table and CMAJOR are then used to set to on the MVX coordinate bit in the statement number text entry for each block from which the variable is busy-on-exit. This procedure is repeated until all variables have been processed. Control is then returned to the LPSEL-IEKPLS subroutine.

To convert not busy-on-entry information to busy-on-entry information, subroutine BIZX-IEKPZ starts with the second MVD table entry, which contains a pointer to the variable assigned coordinate number two, and works down the chain of text blocks. The associated MVX coordinate bit in the statement number text entry for each block is examined. If the coordinate bit is off, the corresponding MVF coordinate bit is inspected. If the MVF coordinate bit is on, a pointer to the associated text block is placed into the busy-on-entry table. This defines the variable to be busy-on-entry to the block (i.e., the variable is used in the block before it is defined). If the associated MVX coordinate bit is on, indicating that the variable is not busy-on-entry, subroutine BIZX-IEKPZ sets the bit to off and proceeds to the next block. This process is repeated until the last text block has been processed.

After the BIZX-IEKPZ subroutine has set to off the MVX coordinate bit (associated with the variable under consideration) in each statement number text entry and built a table of pointers to blocks to which the variable is busy-on-entry, it determines the blocks from which the variable is busy-on-exit.

Starting with the first entry in the busy-on-entry table, subroutine BIZX-IEKPZ obtains (from CMAJOR) pointers to all blocks that are backward connection paths of that entry. Each backward-connecting block is examined to determine whether or not it meets one of three criteria, as follows:

- The block contains a definition of the variable (i.e., the variable's MVS coordinate bit is on).
- The variable has already been marked as busy-on-exit from the block.
- The block corresponds to the busy-on-entry table entry being processed.

If the block meets one of these criteria, the variable is busy-on-exit from the block and its associated MVX coordinate bit is set to on. (The backward connection paths of that block are not explored.)

If the backward-connecting block does not meet any one of these criteria, the variable is marked as busy-on-exit from that block and that block's backward connection paths are, in turn, explored. The same criteria are then applied to the backward-connecting blocks. The backward connection paths are explored in this manner until a block in every path satisfies one of the criteria.

If, during the examination of the backward connection paths, an entry block (i.e., a block lacking backward connection paths) is encountered, the blocks in the table of exit blocks, which was previously built by subroutine BIZX-IEKPZ are used as the backward connection paths for the entry block. Processing then continues in the normal fashion.

When blocks in all backward connection paths have satisfied one of the criteria, the BIZX-IEKPZ subroutine obtains the next entry in the busy-on-entry table and repeats the process. This continues until the busy-on-entry table has been exhausted.

When the busy-on-entry table has been exhausted, the procedure of building the busy-on-entry table and converting it to busy-on-exit information is repeated for the next MVD table entry. When all MVD table entries have been processed, subroutine BIZX-IEKPZ passes control to the LPSEL-IEKPLS subroutine, which calls the loop selection routines.

STRUCTURED SOURCE PROGRAM LISTING

If both the EDIT option and OPT=2 optimization are selected, after subroutine BIZX-IEKPZ has compiled the busy-on-exit information, control is passed to subroutine SRPRIZ-IEKQAA, which records on the SYSPRINT data set a structured source program listing. This listing indicates the loop structure and logical continuity of the source program. (A complete description of the structured source listing is given in the publication IBM System/360 Operating System: FORTRAN IV (G and H) Programmer's Guide, Form C28-6817.)

To produce the listing, subroutine SRPRIZ-IEKQAA reads the SYSUT1 data set prepared by phase 10 and associates, by means of statement numbers, the individual source statements with the text blocks formed from them. By analysis of the loop number information gathered for the text blocks, the SRPRIZ-IEKQAA subroutine then identifies the source statements that make up a particular loop and flags them on the listing by corresponding loop number. Subroutine SRPRIZ-IEKQAA also uses the previously gathered back dominator information to compute listing indentions for the statements. The indentions show dominance relationships; that is, subroutine SRPRIZ-IEKQAA indents the statements that form a text block from the statements that form the back dominator of that block.

LOOP SELECTION

The phase 20 loop selection routine (TARGET-IEKPT) selects the loop to be processed and provides the text optimization and full register assignment routines with the information required to process the loop.

The loop to be processed is selected according to the value of a loop number parameter, which is passed to the loop selection routine. The phase 20 control routine (LPSEL-IEKPLS) sets this parameter to one after the process of structural determination is complete. The TARGET-IEKPT routine is called to select the loop whose blocks have a corresponding loop number. The selected loop is then

passed to the text optimization routines. When text optimization for the loop is completed, the control routine increments the parameter by one, sets the loop number of the blocks in the loop just processed to that of their back target, and marks those blocks as completed. The LPSEL-IEKPLS routine again calls the TARGET-IEKPT routine, which selects the loop whose blocks correspond to the new value of the parameter. The selected loop is then passed to the text optimization routines. This process is repeated until the outermost loop has been text-optimized.

After text optimization has processed the entire module (i.e., the last loop), the control routine removes the block completion marks, initializes the loop number parameter to 1, and passes control to the TARGET-IEKPT routine to reselect the first loop. Control is then passed to the full register assignment routines. When full register assignment for the loop is completed, the control routine marks the blocks of the loop as completed. It then increments the parameter by 1 and passes control to the TARGET-IEKPT routine to select the next loop. Full register assignment is then carried out on the loop. This process is repeated until the outermost loop has undergone full register assignment. (When full register assignment has been carried out on the outermost loop, the LPSEL-IEKPLS routine passes control to the routine that computes the size of each text block and also the displacements required for branching optimization.)

The TARGET-IEKPT routine uses the value of the loop number parameter as a basis for selecting the loop to be processed. The TARGET-IEKPT routine compares the loop number assigned to each text block to the parameter. It marks each block having a loop number corresponding to the value of the parameter as an element of the loop to be processed. It does this by setting on a bit in the block status field of the statement number entry for the block (see Appendix A, "Statement Number/Array Table"). When all such blocks are marked, the loop has been selected.

The information required by the text optimization and full register assignment routines to process the loop consists of the following:

- A pointer to the back target of the loop (if any).
- Pointers to both the first and last blocks of the loop.
- The loop composite matrixes.

After the loop has been selected, this required information is gathered.

Pointer to Back Target

The text optimization and full register assignment routines place both relocated and generated text entries into the back target of the loop. Although the back target of the loop was previously identified during structural determination, it was not saved. Therefore, its identity must be determined again.

The TARGET-IEKPT routine determines the back target of the loop by obtaining the first block of the selected loop. It then analyzes the blocks in the chain of back dominators of the first block to locate the nearest block in the chain that is outside the loop and that passed control to only one block. That block is the back target of the loop, and the TARGET-IEKPT routine saves a pointer to it for use in the subsequent processing of the loop.

Pointers to First and Last Blocks

The pointers to the first and last blocks of the selected loop indicate to the text optimization and full register assignment routines where they are to initiate and terminate their processing. To make these pointers available, the TARGET-IEKPT routine merely determines the first and last blocks of the selected loop and saves pointers to them for use in the subsequent processing of the loop. To determine the first and last blocks, the TARGET-IEKPT routine searches the statement number chain for the first and last entries having the current loop number. The blocks associated with those entries are the first and last in the loop.

Loop Composite Matrixes

The loop composite matrixes, LMVS, LMVF, and LMVX, provide the text optimization and full register assignment routines with a summary of which operands are defined within the selected loop, which operands are used within that loop, and which operands are busy-on-exit from that loop. (An operand is busy-on-exit from the loop if it is used before it is defined in any path along which control flows from the loop.)

The LMVS matrix indicates which operands are defined within the loop. The TARGET-IEKPT routine forms LMVS by combining, via an OR operation, the individual MVS fields in the statement number text entry of every block in the selected loop.

The LMVF matrix indicates which operands are used within the loop. The TARGET-IEKPT routine forms it by combining, via an OR operation, the individual MVF fields in the statement number text entry of every block in the selected loop.

The LMVX matrix indicates which operands are busy-on-exit from the selected loop. LMVX is formed by the TARGET-IEKPT routine. It examines the text entries of each block that is not in the selected loop and that receives control from a block in that loop. Any operand in the text entries of such a block that is either used but not defined in the block or used before it is defined is busy-on-exit from the loop. The TARGET-IEKPT routine sets to on the bit in the LMVX matrix that corresponds to the coordinate assigned to each such operand to reflect that it (i.e., the operand) is busy-on-exit from the loop.

TEXT OPTIMIZATION -- OPT=2

The text optimization process of phase 20 detects text entries within the loop under consideration that do not contribute to the loop's successful execution. These non-essential text entries are either completely eliminated or are relocated to a block outside of the current loop. Because the most deeply nested loops are presented for optimization first, the number of text entries in the most strategic sections of the object module will approach a minimum.

The processing of text optimization is divided into three logical sections:

- Common expression elimination optimizes the execution of a loop by eliminating unnecessary recomputations of identical arithmetic expressions.
- Backward movement optimizes the execution of a loop by relocating to the back target computations essential to the module but not essential to the current loop.
- Strength reduction optimizes the incrementation of DO indexes and the computation of subscripts within the current loop. Modification of the DO increment may allow multiplications to be relocated into the back target. If the DO increment is not busy-on-exit from the loop, it may be completely replaced by a new DO increment that becomes both a subscript value and a test value at the bottom of the DO loop.

The first two of the foregoing sections are similar in that they examine text entries in strict order of occurrence within the loop.

The last section does not examine individual text entries within the loop; instead, the TYPES table, constructed prior to their execution, is consulted for optimization possibilities. Furthermore, an interaction of entries in the TYPES table must exist before processing can proceed. The TYPES table contains pointers to type 3, 5, 6, and 7 text entries. The various types, their definitions, and the section(s) of text optimization that process them are outlined in Table 4. Pointers to type 1 and type 2 text entries are not entered into the TYPES table. The reason is that such types have already been processed during backward movement.

The following text describes the processing performed by each of the sections of the text optimization. Table 11 summarizes the criteria for performing text optimization in each section. An example illustrating the type of processing of each section is given in Appendix D. These examples should be referred to when reading the text describing the processing of the sections.

Table 4. Text Entry Types

Type	Definition	Processed by
Type 1	A text entry having an absolute constant[1] in both the operand 2 and operand 3 position.	Backward Movement (elimination)
Type 2	A text entry having stored constants[2] in both the operand 2 and operand 3 positions.	Backward Movement (movement)
Type 3	An inert text entry (i.e., a text entry that is a function of itself and an additive constant; e.g., J=J+1).	Strength Reduction
Type 5	A text entry whose operand 1 (a temporary) is a function of a variable (or temporary) and a constant, and whose operator is multiplicative (* or /).	Strength Reduction
Type 6	A text entry whose operand 1 (a temporary) is a function of a variable (or temporary) and a constant, and whose operator is additive (+ or -).	Strength Reduction
Type 7	A branch text entry	Strength Reduction

[1]Absolute constants are those that agree with the definition of numerical constants as stated in the publication IBM System/360 Operating System: FORTRAN IV Language, Form C28-6515.

[2]A stored constant is a variable that is not defined within a loop and, thus, its value remains constant throughout execution of that loop.

Common Expression Elimination -- OPT=2

The object of common expression elimination, which is carried out by subroutine XPELIM-IEKQXM, is to get rid of any unnecessary arithmetic expressions. This is accomplished by eliminating text entries, one at a time, until the entire expression disappears. An arithmetic text entry is unnecessary if it represents a value (calculated elsewhere in the loop) that may be used without modification. A value may be used without modification if, between appearances of the same computation, operands 2 and 3 of the text entry are not redefined. The following paragraphs discuss the processing that occurs during common expression elimination.

Within the current loop, subroutine XPELIM-IEKQXM examines each uncompleted block (i.e., a block that is not part of an inner loop) for text entries that are candidates for elimination. A text entry is a candidate if it contains an arithmetic, binary, logical, or subscript operator. Once a candidate is found, the XPELIM-IEKQXM subroutine attempts to locate a matching text entry. A text entry matches the candidate if operand 2, operand 3, and the operator of that text entry are identical to those of the candidate. If either operand 2 or 3 of the matching text entry is redefined between that text entry and the candidate, the match is not accepted. The search for the matching text entry takes place in the following locations:

- In the same block as the candidate, between the first text entry and the candidate.
- In a back dominator (see note) of the block in which the candidate resides.

  Note: Only back dominators that are not elements of previously processed loops and that are within the confines of the current loop are considered. The first back dominator considered is the one nearest to the block being processed. The next considered is the back dominator of the nearest back dominator, etc.

When a matching text entry is found, subroutine XPELIM-IEKQXM performs elimination in the following way:

- If operand 1 of the matching text entry is not redefined between that text entry and the candidate, subroutine XPELIM-IEKQXM substitutes that operand

for operand 2 of the candidate and converts the operator to a store.

- If, however, operand 1 is redefined, subroutine XPELIM-IEKQXM generates a text entry to save the value of operand 1 in a temporary and inserts this text entry into text immediately after the matching text entry. It then replaces operand 2 of the candidate with this temporary, and converts the operator to a store.

- Finally, if operand 1 of the candidate is a temporary generated by phase 15, the XPELIM-IEKQXM subroutine replaces all uses of the temporary with the new operand 2 of the candidate and deletes the candidate. Thus, the value of the matching text entry is propagated forward for a possible match with another candidate. This provides the link to the next text item of the complete common expression.

All text entries in the block under consideration are processed in the previously described manner. When the entire block is processed, the next uncompleted block in the loop is selected and its text entries undergo common expression elimination. When all uncompleted blocks in the loop are processed, control is returned to the phase 20 control routine, which passes control to the portion of phase 20 that continues text optimization through backward movement.

The overall logic of common expression elimination is illustrated in Chart 11. An example of common expression elimination is given in Appendix D.

Backward Movement -- OPT=2

Backward movement, which is performed by subroutine BACMOV-IEKQBM, moves text entries from a loop to an area that is executed less often, the back target of the loop. During backward movement, each uncompleted block in the loop being processed is examined for text entries that are candidates for backward movement. To be a candidate for backward movement, a text entry must be type 2. Therefore, it must:

- Contain an arithmetic or logical operator.

- Have operands 2 and 3 that are not defined within the loop.

When a candidate is found, subroutine BACMOV-IEKQBM carries out backward movement of that candidate in one of two ways:

- If operand 1 of the candidate is not busy-on-exit from the back target of the loop and if operand 1 of the candidate is not defined elsewhere in the loop, the BACMOV-IEKQBM subroutine moves the entire candidate to the back target of the loop. (An operand is not busy-on-exit from the back target if that operand is defined in the loop before it is used.)

- If operand 1 of the candidate is busy-on-exit from the back target of the loop or if it is defined elsewhere in the loop, subroutine BACMOV-IEKQBM generates a text entry to perform the computation of the expression in the candidate and store the result in a new temporary. It moves this text entry to the end of the back target of the loop and then replaces the expression in the candidate with operand 1, the new temporary, of the generated text entry.

All the text entries in the block under consideration are processed in the previously described manner. When the entire block is processed, the next uncompleted block in the loop is selected and its text entries undergo backward movement. When all uncompleted blocks in the loop are processed, control is returned to the phase 20 control routine, which passes control to the portion of phase 20 that continues text optimization through strength reduction.

The overall logic of backward movement is illustrated in Chart 12. An example of backward movement is given in Appendix D.

Two additional optimization processes are performed concurrently with backward movement. They are the elimination of simple stores and of arithmetic expressions that appear in text entries and are functions of constants.

Elimination of Simple Stores: The BACMOV-IEKQBM subroutine effects the removal of unnecessary simple stores (i.e., text entries of the form "operand 1 = operand 2") from the block that is currently undergoing backward movement. The following paragraph describes the processing.

Subroutine BACMOV-IEKQBM selects as candidates for elimination any simple store in which operand 1 is a nonsubscripted variable. Pointers to the candidates are passed to the SUBSUM-IEKQSM subroutine, which determines if elimination is indeed possible according to the conditions

illustrated in Table 5. At the same time, subroutine SUBSUM-IEKQSM replaces all uses of operand 1 of the candidate with operand 2 of the candidate in text entries between either:

- The candidate and the first redefinition of either operand.
- The candidate and the end of the block.

The BACMOV-IEKQBM subroutine then deletes those candidates so marked by subroutine SUBSUM-IEKQSM. An example of simple-store elimination is illustrated in Appendix D.

Table 5. Operand Characteristics That Permit Simple-Store Elimination

Operand 1 Busy-on-Exit from Block	Operand 1 Redefined Later in Block	Operand 2 Redefined Before Operand 1 Redefined	Operand 1 Used After Operand 2 Redefined
No	No	No	X
No	No	Yes	No
No	Yes	No	X
No	Yes	Yes	No
Yes	Yes	No	X
Yes	Yes	Yes	No

X = condition cannot exist because of previous characteristics of operands.

Elimination of Text Entry Expressions Involving Integer Constants (Type 1): During the scan of a block for text entries to be moved to the back target, subroutine BACMOV-IEKQBM also checks for text entries whose operators are arithmetic and whose operands 2 and 3 are both integer constants. When such a text entry is found, the BACMOV-IEKQBM subroutine eliminates the arithmetic expression in the text entry by:

- Calculating the result of the expression.
- Creating a new dictionary entry for the result, which is a constant.
- Replacing the arithmetic expression with the result.

The text entry is thereby reduced to a simple store, which may be eliminated by simple-store elimination.

Strength Reduction -- OPT=2

Strength reduction, which is performed by subroutine REDUCE-IEKQSR, optimizes loops that are controlled by logical IF statements. (DO loops are converted to loops controlled by logical IF statements during phase 10 processing.) Such loops are optimized by modifying the expression (e.g., $J \leq 20$) in the IF statement; this enables certain text entries to be moved from the loop to the back target of the loop, an area executed less frequently. Strength reduction processing is divided into two sections:

- Elimination of multiplicative text.
- Elimination of additive text.

Both of these sections perform strength reduction, but each has a separate set of criteria for considering a loop as a candidate for reduction. However, the manner in which each section implements reduction essentially is the same.

Elimination of Multiplicative Text: To eliminate multiplicative text, subroutine REDUCE-IEKQSR examines the loop being processed to determine whether or not it is a candidate for strength reduction. The loop is a candidate if:

- The loop contains an inert text entry (type 3).
- Operand 1 of the inert text entry is used in another text entry (in the loop) whose operator indicates multiplication and whose other used operand is a constant[1] (type 5).
- Operand 1 of the inert text entry is the variable appearing in the expression of the logical IF statement that controls the loop.

If the loop is a candidate, subroutine REDUCE-IEKQSR implements strength reduction in one of two ways:

1. If the constants in the inert text entry and the multiplicative text entry are both absolute constants, the REDUCE-IEKQSR subroutine:
   a. Calculates a new constant (K) equal to the product of the absolute constants.

---

[1]This other text entry is referred to as a multiplicative text entry.

b. Generates another inert text entry and inserts it into the loop immediately after the original inert text entry. The additive constant in this text entry is K.

c. Modifies the expression in the logical IF statement by:

   (1) Replacing the branch variable (see note) with operand 1 of the generated inert text entry.

   (2) Replacing the branch constant (see note) with a constant equal to the product of the branch constant and the absolute constant in the multiplicative text entry.

d. Deletes the original inert text entry if operand 1 of that text entry is not busy-on-exit from the loop.

e. Moves the multiplicative text entry to the back target of the loop.

f. Replaces operand 1 of the multiplicative text entry with operand 1 of the generated inert text entry.

g. Replaces the uses of operand 1 of the multiplicative text entry that remain in the loop with operand 1 of the generated inert text entry.

   Note: The branch variable is the variable in the expression of the logical IF statement that is tested to determine whether or not the loop is to be re-executed. The branch constant is the constant with which the branch variable is compared. For example, in IF (J ≤ 3) where J is the branch variable and 3 is the branch constant.

2. If either of the constants in the inert text entry or the multiplicative text entry is a stored constant, the REDUCE-IEKQSR subroutine performs similar processing to that described above. However, prior to generating the inert text entry, it generates an additional text entry and places it into the back target of the loop. This text entry multiplies the two constants. Operand 1 of this text entry becomes the additive constant in the generated inert text entry. In the case where the constant in the multiplicative text entry is a stored constant, a second additional text entry is generated and placed into the back target of the loop. This second text entry multiplies the branch constant by the constant in the multiplicative text entry. Operand 1 of the second text entry becomes the new branch constant of the logical IF.

If additional multiplicative text entries exist within the loop, the foregoing process is repeated. Repetitive processing of this type results in a number of generated inert text entries, which may be eliminated from the loop by the processing of the second section of strength reduction.

Elimination of Additive Text: To eliminate additive text, subroutine REDUCE-IEKQSR examines the loop being processed to determine whether or not it is a candidate for strength reduction. The loop is a candidate if:

- The loop contains an inert text entry (type 3).
- Operand 1 of the inert text entry is used in the loop in another text entry whose operator indicates addition[1] (type 6).

If the loop is a candidate, the processing performed by subroutine REDUCE-IEKQSR to eliminate the additive text entry is essentially the same as that performed to eliminate a multiplicative text entry.

The overall logic of strength reduction is illustrated in Chart 13. An example showing both methods of strength reduction is given in Appendix D.

FULL REGISTER ASSIGNMENT -- OPT=2 (CHART 14)

During OPT=2 optimization, full register assignment is carried out on module loops, rather than on the entire module, as is the case for OPT=1 optimization. Regardless of whether a loop or the entire module is being processed, the full register assignment routines operate essentially in the same manner. However, the optimization effect of full register assignment, when carried out on a loop-by-loop basis, is more pronounced. Because the most deeply nested loops are presented for full register assignment first, the number of

---

[1]This text entry is referred to as an additive text entry.

register loads in the most strategic sections ofthe object module approaches a minimum. The processing of a loop by full register assignment differs from the processing of the entire module only in the area of global assignment. An understanding of the processing performed on a loop, other than global assignment, can be derived from the previous discussion of full register assignment (see "Full Register Assignment -- OPT=1"). Global assignment for a loop is described in the following text.

When processing a loop, the global assignment routine (GLOBAS-IEKRGB) incorporates into the current loop, wherever possible, the global assignments made to items (i.e., operands and base addresses) in previously processed loops. It does this to ensure that the same register is assigned in both loops if an item eligible for global assignment in the current loop was globally assigned in a previously processed loop.

Before the global assignment routine assigns an available register to the most active item of the current loop, it determines whether that item was globally assigned in a previously processed loop. (As global assignment is carried out on each loop, all global assignments for that loop are recorded and saved for use when the next loop is considered.) If the item was not globally assigned in a previously processed loop, the GLOBAS-IEKRGB routine assigns it the first available register. If the item was globally assigned in a previously processed loop, the global assignment routine then determines whether or not the register assigned to the item in the previously processed loop is currently available. If that register is available, the GLOBAS-IEKRGB routine also globally assigns it to the same item in the current loop. If the register is not available, the global assignment of that item in the previously processed loop cannot be incorporated into the current loop. The GLOBAS-IEKRGB routine, therefore, assigns the item an available register different from that assigned to it in the previously processed loop. The GLOBAS-IEKRGB routine selects the eligible item with the next highest activity in the current loop and treats it in the same manner. Processing continues in this fashion until the supply of eligible items or the supply of available registers is exhausted.

As each global assignment is made to an active item, the GLOBAS-IEKRGB routine checks to determine whether or not that item is busy-on-exit from the back target of the loop. If the item is busy-on-exit, the GLOBAS-IEKRGB routine generates a text entry to load that item into the assigned register and inserts it into the back target of the loop. The load is required to guarantee that the item is in a register and available for subsequent use during loop execution. If the item is not busy-on-exit, the text item is not required to be loaded. If any globally assigned item is defined within the loop and is also busy-on-exit from the loop, the GLOBAS-IEKRGB routine generates a text entry to store that item on exit from the loop. The generated store is needed to preserve the value of such an operand for use when it is required during the execution of an outer loop.

The GLOBAS-IEKRGB routine records all global assignments made for the current loop for use in the subsequent updating scan (see "Full Register Assignment -- OPT=1") and also for incorporation, wherever possible, into subsequently processed loops.

BRANCHING OPTIMIZATION -- OPT=2

During OPT=2 optimization, branching optimization is carried out in the same manner as during OPT=1 optimization. After all loops have undergone full register assignment, subroutine BLS-IEKSBS is given control to calculate the size of each block. When the sizes of all blocks have been calculated, the BLS-IEKSBS subroutine uses the block size information to determine the blocks to which a branch can be made by means of RX-format branch instructions.

PHASE 25

Phase 25 completes the production of an object module from the combined output of the preceding phases of the compiler. An object module consists of four elements:

- Text information.
- External symbol dictionary.
- Relocation dictionary.
- Loader END record.

The text information (instructions and data resulting from the compilation) is in a relocatable machine language format. It may contain unresolved external symbolic cross references (i.e., references to symbols that do not appear in the object module). The external symbol dictionary contains the information needed to resolve the external symbolic cross references that appear in the text information. The

relocation dictionary contains the information needed to relocate the text information for execution. The END record informs the linkage editor of the length of the object module and the address of its main entry point.

An object module resulting from a compilation consists of a single control section, unless common blocks are associated with the module. An additional control section is included in the module for each common block.

The object module produced by phase 25 is recorded on the SYSLIN data set if the LOAD option is specified by the FORTRAN programmer, and on the SYSPUNCH data set if the DECK option is specified. If the LIST option is specified, phase 25 develops and records on the SYSRINT data set a pseudo-assembler language listing of the instructions and data of the object module. If the MAP option is specified, phase 25 also produces a storage map. If the ID option is specified, phase 25 inserts information into the object module which is used by the object-time traceback routine of the Library.

TEXT INFORMATION

Text information consists of the machine language instructions and data resulting from the compilation. Each text information entry (a TXT record) constructed by phase 25 can contain up to 56 bytes of instructions and data, the address of the instructions and data relative to the beginning of the control section, and an indication of the control section that contains them. A more detailed discussion of the use and format of TXT records is given in the publication IBM System/360 Operating System: Linkage Editor, Program Logic Manual, Form Y28-6610.

The major portion of phase 25 processing is concerned with text information construction. In building text information, phase 25 obtains each item that is to be placed into text information, converts the item to machine language format wherever necessary, enters the item into a TXT record, and places the relative address of the item into the TXT record.

Phase 25 assigns relative addresses by means of a location counter, which is continually updated to reflect the location at which the next item is to be placed into text information. Whenever phase 25 begins the construction of a new TXT record, it inserts the current value of the location counter into the address field of the TXT record. Thus, the address field of the TXT record indicates the relative address of the instructions and data that are placed into the record.

Figure 9 shows the layout of storage that phase 25 assumes in setting up text information.

Phase 25 constructs text information by:

- Reserving address constants for the referenced statement numbers of the module.
- Completing the processing of the adcon table entries and entering the resultant entries into TXT records.
- Generating the prologue and epilogue instructions and entering these instructions into TXT records.
- Converting phase 15/phase 20 text into System/360 machine code and entering the code into TXT records.

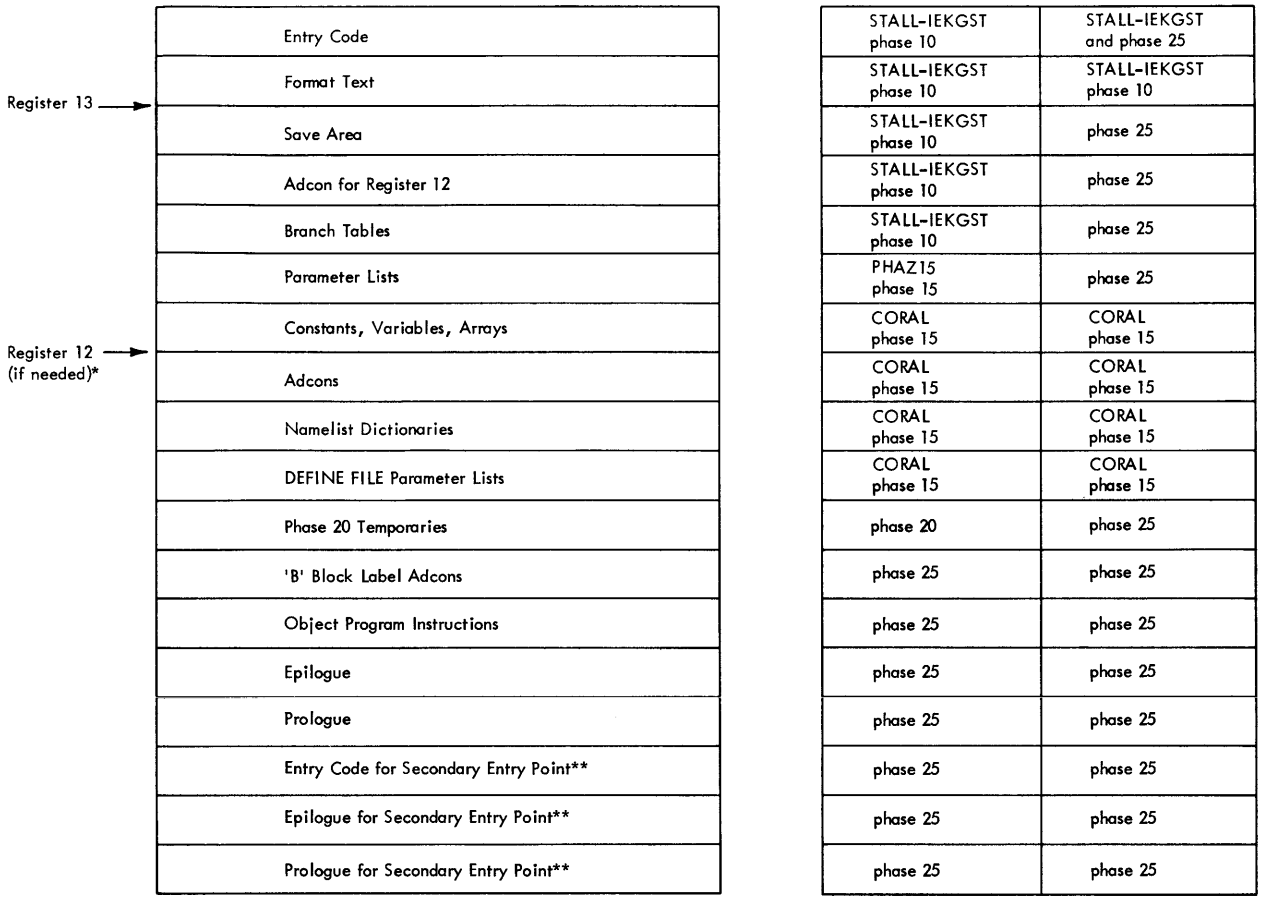
Chart 20 shows the overall logic of phase 25 processing.

Address Constant Reservation

Before it constructs text information, subroutine MAINGN-IEKTA reserves address constants for the referenced statement numbers of the module and for the statement numbers appearing in computed GO TO statements. The address constants are reserved so that the relative addresses of the statements associated with such statement numbers can be recorded and, subsequently, obtained during execution of the object module, when branches to those statements are required.

To reserve address constants for statement numbers, subroutine MAINGN-IEKTA scans the chain of statement number entries in the statement number/array table. For each encountered statement number to which reference is made, subroutine MAINGN-IEKTA inserts a base and displacement into the associated statement number entry. When the text representation of that statement number is encountered, a relative address is placed in the statement number entry.

Note: If branching optimization is being implemented, subroutine MAINGN-IEKTA does not perform the processing described in the previous paragraph.



*See "Relative Address Assignment" under "CORAL Processing."
**See last paragraph of "Generation of Initialization Instructions" under "FORTRAN System Director."

Figure 9. Storage Layout for Text Information Construction

After all statement numbers are processed, bases and displacements are likewise assigned to adcons for the statement numbers appearing in computed GO TO statements. The MAINGN-IEKTA subroutine scans the branch table chain (see Appendix A, "Branch Tables"), and assigns a base and displacement for each branch table. Subroutine MAINGN-IEKTA does not record pointers to the address constants set aside for the actual statement numbers of the computed GO TO statements in their associated standard branch table entries. The values to be placed into the address constants for statement numbers in computed GO TO statements are also determined during text conversion.

Text Conversion

Phase 25 converts intermediate text into System/360 machine code. (The text conversion process is controlled by subroutine MAINGN-IEKTA.) In converting the text, phase 25 obtains each text entry and, depending upon the nature of the operator in the text entry, passes control to one of six processing paths to convert the text entry.

The six processing paths are:

- Statement Number Processing.
- Input/output Statement Processing.
- CALL Statement Processing.
- Code Generation.
- RETURN Statement Processing.
- END Statement Processing.

See Table 14 for the complete list of subroutines called by phase 25.

STATEMENT NUMBER PROCESSING: When the operator of the text entry indicates a statement number, subroutine MAINGN-IEKTA passes control to subroutine LABEL-IEKTLB. The LABEL-IEKTLB subroutine then inserts

the current value of the location counter, which is the relative address of the statement associated with the statement number, into the statement number entry. All branches to that statement are made through the use of the relative address for that statement number.

Note: If branching optimization is being implemented, only statement numbers to which a branch cannot be made via RX-format branch instructions (i.e., statement numbers that are not within the range of registers 13, 12, 11, 10, and 9) are processed as described above.

After the relative address has been placed into the statement number entry, subroutine LABEL-IEKTLB determines whether or not that statement number appears in a computed GO TO statement. If it does, subroutine LABEL-IEKTLB also inserts the relative address into the appropriate field of the branch table entry, or entries, for that statement number. The relative address recorded in the branch table entry is placed into the storage reserved for it within text information (see "END Statement Processing") when the text representation of the END statement is encountered.

INPUT/OUTPUT STATEMENT PROCESSING: When the operator of the text entry indicates an input/output statement, an I/O list item, or the end of an I/O list, the MAINGN-IEKTA subroutine passes control to subroutine IOSUB-IEKTIS, which generates an appropriate calling sequence to IHCFCOMH to perform, at object-time, the indicated operation.

The calling sequence generated for an input/output statement depends on the type of the statement (e.g., READ, BACKSPACE). The calling sequence generated for an I/O list item depends on the input/output statement type with which the list item is associated and on the nature of the list item, i.e., whether the item is a variable or an array. The calling sequence generated for an end of an I/O list depends on whether the end I/O list operator signals:

- The end of an I/O list associated with a READ/WRITE that requires a FORMAT statement.
- The end of an I/O list associated with a READ/WRITE that does not require a FORMAT statement.

Once the calling sequence is generated, subroutine IOSUB-IEKTIS enters it into TXT records.

CALL STATEMENT PROCESSING: When the operator of the text entry indicates a CALL statement, subroutine MAINGN-IEKTA passes control to subroutine FNCALL-IEKVFN to generate a standard direct-linkage calling sequence, which uses general register 1 as the argument register. The argument list is located in the adcon table in the form of address constants. Each address constant for an argument contains the relative address of the argument. The FNCALL-IEKVFN subroutine enters the calling sequence into TXT records.

CODE GENERATION: Code generation converts text entries having operators other than those for statement numbers, ENTRY, CALL, RETURN, END, and input/output statements into System/360 machine code. To convert the text entry, code generation uses four arrays and the information in the text entry. The four arrays are:

- Register array. This array is reserved for register and displacement information.
- Directory array. This array contains pointers to the skeleton arrays and the bit-strip arrays associated with operators in text entries that undergo code generation.
- Skeleton array. A skeleton array exists for each type of operator in an intermediate text entry that is to be processed by code generation. The skeleton array for a particular operator consists of all the machine code instructions, in skeleton form and in proper sequence, needed to convert the text entry containing the operator into machine code. These instructions are used in various combinations to produce the desired object code. (The skeleton arrays are shown in Appendix C.)
- Bit-strip array. A bit-strip array exists for each type of operator in a text entry that is to undergo code generation. One strip is selected for each conversion involving the operator. The bits in each strip are preset (either on or off) in such a fashion that when the strip is matched against the skeleton array, the strip indicates the combination of instructions that is to be used to convert the text entry. (The bit strip arrays are shown with their associated skeleton arrays in Appendix C.)

In code generation, the actual base registers and operational registers (i.e., registers in which calculations are to be performed), assigned by phase 20 to the operands of the text entry to be converted

to machine code, are obtained from the textentry and placed into the register array. Any displacements needed to load the base addresses of the operands are also placed into the register array. The displacements referred to in this context are the displacements of the base addresses of the operands from the start of the adcon table that contains the base addresses. These displacements are obtained from the information table entries for the operands. This action is taken to facilitate subsequent processing.

The operator of the text entry to be converted is used as an index to the directory array. The entry in this directory array, which is pointed to by the operator index, contains pointers to the skeleton array and the bit-strip array associated with the operator.

The proper bit strip is then selected from the bit-strip array. The selection depends on the status of operand 2 and operand 3 of the text entry. This status is set up by phase 20 and is indicated in the text entry by four bits (see Appendix A, "Phase 20 Intermediate Text Modifications"): the first two bits indicate the status of operand 2; the second two bits indicate the status of operand 3.

The status of operand 2 and/or operand 3 can be one of the following:

00 The operand is in main storage and is to remain there after the present code generation. Therefore, if the operand is loaded into a register during the present code generation, the contents of the register can be destroyed without concern for the operand.

01 The operand is in main storage and is to be loaded into a register. The operand is to remain in that register for a subsequent code generation; therefore, the contents of the register are not to be destroyed.

10 The operand is in a register as a result of a previous code generation. After the register is used in the present code generation process, its contents can be destroyed.

11 The operand is in a register and is to remain in that register for a subsequent code generation. The contents of the register are not to be destroyed.

This four-bit status field is used as an index to select a bit strip from the bit-strip array associated with the operator. The combination of instructions indicated in the bit strip conforms to the operand status requirements: i.e., if the status of operand 2 is 11, the generated instructions make use of the register containing operand 2 and do not destroy its contents. The combination, however, excludes base load instructions and the store into operand 1.

Once the bit strip is selected, it is moved to a work area. The strip is modified to include any required base load instructions. That is, bits are set to on in the appropriate positions of the bit strip in such a way that, when the strip is matched to the skeleton array, the appropriate instructions for loading base addresses are included in the object code. The skeletons for these load instructions are part of the skeleton array.

The code generation process determines whether or not the base address of operand 2 and/or operand 3 must be loaded into a register by examining the status of these base addresses in the text entry. Such status is indicated by four bits: the first two bits indicate the status of the base address of operand 2; the second two bits indicate the status of the base address of operand 3. If this status field indicates that a base address is to be loaded, the appropriate bit in the bit strip is set to on. (The bit to be operated upon is known, because the format of the skeleton array for the operator is known.)

Before the actual match of the bit strip to the skeleton array takes place, the code generation process determines:

- If the base address of operand 1 must be loaded into a register.
- If the result produced by the actual machine code for the text entry is to be stored into operand 1.

This information is again indicated in the text entry by four bits: the first two bits indicate the status of the base address of operand 1; the second two bits indicate whether or not a store into operand 1 is to be included as part of the object code. If the base address of operand 1 is to be loaded and/or if operand 1 is to be stored into, the appropriate bit(s) in the bit strip is set to on.

The bit strip is then matched against the skeleton array. Each skeleton instruction corresponding to a bit that is

set to on in the bit strip is obtained and converted to actual machine code. The operation code of the skeleton instruction is modified, if necessary, to agree with the mode of the operand of the instruction. The mode of the operand is indicated in the text entry. The symbolic base, index, and operational registers of the skeleton instructions are replaced by actual registers. The base and operational registers to be used are contained in the register array. If an operand is to be indexed, the index register to be used is obtained. (The index register is saved during the processing of the text entry whose third operand represents the actual index value to be used.) The displacement of the operand from its base address, if needed, is obtained from the information table entry for the operand. (The contents of the displacement field of the text entry are added to this displacement if a subscript text entry is being processed.) These elements are then combined into a machine instruction, which is entered into a TXT record. (If the skeleton instruction that is being converted to machine code is a base load instruction, the base address of the operand is obtained from the object-time adcon table. The register (12) containing the address of the adcon table and the displacement of the operand's base address from the beginning of the adcon table are contained in the register array.)

Branch Processing: The code generation portion of phase 25 generates the machine code instructions to complete branching optimization. The processing performed by code generation, if branching optimization is being implemented, is essentially the same as that performed to produce an object module in which branching is not optimized. However, before a skeleton instruction (corresponding to an on bit in the selected and modified bit strip) is assembled into a machine code instruction, code generation determines whether or not that instruction:

- Loads into a register the address of an instruction to which a branch is to be made and which is displaced less than 4096 bytes from the address in a reserved register.[1]

- Is an RR-format branch instruction that branches to an instruction that is displaced less than 4096 bytes from the address in a reserved register.[2]

Note: A load candidate usually immediately precedes a branch candidate in the skeleton array.

Code generation determines whether or not the instruction to which a branch is to be made is displaced less than 4096 bytes from an address in a reserved register by interrogating an indicator in the statement number entry for the statement number associated with the block containing the instruction to which a branch must be made. This indicator is set by phase 20 to reflect whether or not that block is displaced less than 4096 bytes from an address in a reserved register.

The completion of branching optimization proceeds in the following manner. If a skeleton instruction corresponding to an on bit in the bit strip is a load condidate, it is not included as part of the instruction sequence generated for the text entry under consideration. If a skeleton instruction corresponding to an on bit in the bit strip is a branch candidate, it is converted to an RX-format branch instruction. The conversion is accomplished by replacing operand 2 (a register) of the branch candidate with an actual storage address of the format D(0,Br). D represents the displacement of the instruction (to which a branch is to be made) from the address that is in the appropriate reserved register (Br).

If the instruction to which a branch is to be made is the first in the text block, both the displacement and the reserved register to be used for the RX-format branch are obtained from the statement number entry associated with the block containing the instruction. (This information is placed into the statement number entry during phase 20 processing.)

If the instruction to which a branch is to be made is one that is subsequently to be included as part of the instruction sequence generated for the text entry under consideration,[3] the displacement of the instruction from the address in the appropriate reserved register is computed and used as the displacement of the RX-format branch instruction. The reserved register used in such a case is the one indicated in the statement number entry associated with the block containing the text entry currently being processed by code generation.

---

[1]This type of text entry is subsequently referred to as a load candidate.

[2]This type of text entry is subsequently referred to as a branch candidate.

[3]Skeleton arrays for certain operators contain RR format branch instructions that transfer control to other instructions of that skeleton.

RETURN STATEMENT PROCESSING: When the operator of the text entry indicates a RETURN statement, subroutine MAINGN-IEKTA passes control to subroutine RETURN-IEKTRN, which generates a branch to the epilogue. The epilogue address is obtained from the save area. The address of the epilogue is placed into the save area during the execution of either the subprogram main entry coding or the subprogram secondary entry coding. The address of the epilogue is placed into the save area during the compilation of a main program or subprogram with no secondary entry points (refer to the section "Initialization Instructions").

END STATEMENT PROCESSING (CHART 21): When the operator of the text entry indicates an END statement, subroutine MAINGN-IEKTA passes control to subroutine END-IEKUEN, which completes the processing of the module by entering the address constants (i.e., relative addresses) for statement numbers and statement numbers appearing in computed GO TO statements into text information and by generating the END record.

Subroutine END-IEKUEN calls the ENTRY-IEKTEN subroutine to determine whether or not the program being compiled is a main program or a subprogram and to take the appropriate action. If it is a subprogram, the ENTRY-IEKTEN subroutine calls subroutine EPILOG-IEKTEP and PROLOG-IEKTPR (see "Prologue and Epilogue Generation"). If it is a main program, subroutine ENTRY-IEKTEN generates code to call IHCFCOMH and generates a branch to the appropriate place in text. If there are secondary entry points, text is scanned to determine where they are located. An epilogue and prologue are generated for each entry point with a branch to the corresponding point in the object code. Subroutine ENTRY-IEKTEN returns control to the END-IEKUEN subroutine.

Subroutine END-IEKUEN places TXT and RLD records in the object module for the following: adcon for the save area, adcon for the prologue, adcon for the epilogue, adcon for register 12 (if needed), adcons for branch tables, adcons for parameter lists, and adcons for 'B' block labels. The END-IEKUEN subroutine generates TXT information for each temporary. Subroutine END-IEKUEN calls IEND (FSD entry point) to generate the loader END record that must be the last record of the object module. Its functions are to signal the end of the object module and to inform the linkage editor of the size (in bytes) of the control section and the address of the main entry point of the control section. The END-IEKUEN subroutine then returns control to the FSD through subroutine MAINGN-IEKTA.

Storage Map Production

As a user option, subroutine IEKGMP produces a storage map of the symbols used in the source program. The map contains the following information:

Name	Symbol	Explanation
Tag	S	The variable appeared to the left of an equal sign in the source program. (stored into)
	F	The variable appeared to the right of an equal sign in the source program. (fetched)
	A	The variable was used as an argument.
	C	The variable appeared in a COMMON statement.
	E	The variable appeared in an EQUIVALENCE statement.
	XR	The variable is a call-by-name parameter to the source program.
	XF	The variable is a subroutine or function name.
	ASF	The variable is the name of an arithmetic statement function.
Type		Identifies the type of variable -- Type * length -- in bytes.
Add.		Is the relative address of the variable within the object module (in hexadecimal).

The total size of the object module is also given.

A map of each COMMON block is generated to give the relative location of each variable in that COMMON block. A map of variables equivalenced into common is also provided.

In addition, subroutine TENTXT-IEKVTN generates a map of statement numbers.

Prologue and Epilogue Generation

Phase 25 generates the machine code: (1) to transmit parameters to a subprogram, and (2) to return control to the calling routine after execution of the subprogram. Parameters are transmitted to the subprogram by means of a prologue. Return

is made to the calling routine by means of an epilogue. Prologues and epilogues are provided for subprogram secondary entry points as well as for the main entry point.

Prologue: A prologue (generated by subroutine PROLOG-IEKTPR) is a series of load and store instructions that transmit the values of "call by value" parameters and the addresses of "call by name" parameters to the subprogram. (These parameters are explained in the publication IBM System/360 Operating System: FORTRAN IV Language, Form C28-6515.)

When subroutine PROLOG-IEKTPR generates a prologue, it enters the prologue into TXT records and inserts its relative address into the address constant reserved for the prologue address during the generation of initialization instructions.

Epilogue: An epilogue (generated by subroutine EPILOG-IEKTEP) is a series of instructions that (1) return to the calling routine the values of "call by value" parameters (if they are stored into or used as arguments), (2) restore the registers of the calling routine, and (3) return control to the calling routine. (If "call by value" parameters do not exist, an epilogue consists of only those instructions required to restore the registers and to return control.)

When subroutine EPILOG-IEKTEP generates an epilogue, it enters the epilogue into TXT records and inserts its relative address into the address constant reserved for the epilogue address during the generation of initialization instructions. (When phase 25 encounters the text representation of a RETURN statement, a branch to the epilogue is generated.)

PHASE 30

Phase 30 records appropriate messages (on the SYSPRINT data set) for syntactical errors encountered during the processing of previous phases; its overall logic is illustrated in Chart 22. (Table 15 shows the subroutines called by phase 30.) As errors are encountered by these phases, error table entries are created and placed into an error table. Each such entry consists of two parts. The first part contains a message number. (If the error cannot be localized to a particular statement, no internal statement number is entered in the error table entry. Phase 30 simulates the internal statement number with a zero.) The second part contains either an internal statement number if the entry is for a statement that is in error, a dictionary pointer to a variable if the entry is for a variable that is in error, or an actual statement number if the entry is for an undefined statement number.

Message Processing

Using the message number in the error table entry multiplied by four, phase 30 locates, within the message pointer table (see Appendix A, "Diagnostic Message Tables"), the entry corresponding to the message number. This message pointer table entry contains (1) the length of the message associated with the message number, and (2) a pointer to the text of the message associated with the message number. After phase 30 obtains the pointer to the message text, it constructs a parameter list, which consists of:

- Either the internal statement number, dictionary pointer, or statement number appearing in the error table entry.
- A pointer to the message text associated with the message number.
- The length of the message.
- The message number.
- The error level.

Having constructed the parameter list, phase 30 calls subroutine MSGWRT-IEKP31, which writes the message on the SYSPRINT data set. After the message is written, the next error table entry is obtained and processed as described above.

As each error table entry is being processed, the error level code (either 4, 8, or 16) associated with the message number is obtained from the error code table (GRAVERR) by using the message number in the error table entry as an index. The error level code indicates the seriousness of the encounter error. (For explanations of all the messages the compiler generates, see the publication IBM System/360 Operating System: FORTRAN IV (G and H) Programmer's Guide, Form C28-6817.) The obtained error level code is saved for subsequent use only if it is greater than the error level codes associated with message numbers appearing in previously processed error table entries. Thus, after all error table entries have been processed, the highest error level code (either 4, 8, or 16) has been saved. The saved error level code is passed to the FSD when phase 30 processing is completed. This code is used as a return code by the scheduler to determine whether or not succeeding steps are to be executed.

Chart 00. Compiler Control Flow

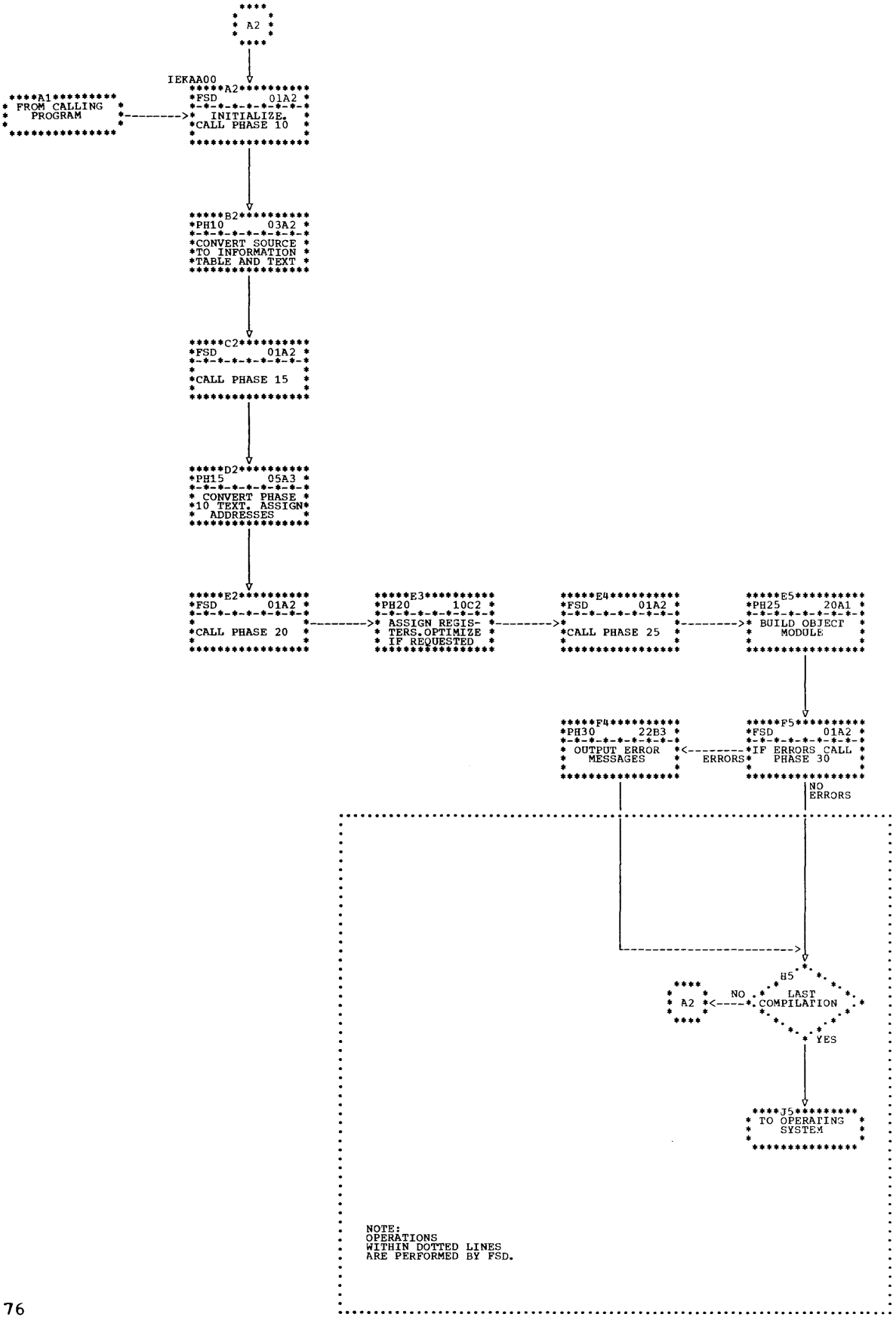


Chart 01. FSD Overall Logic

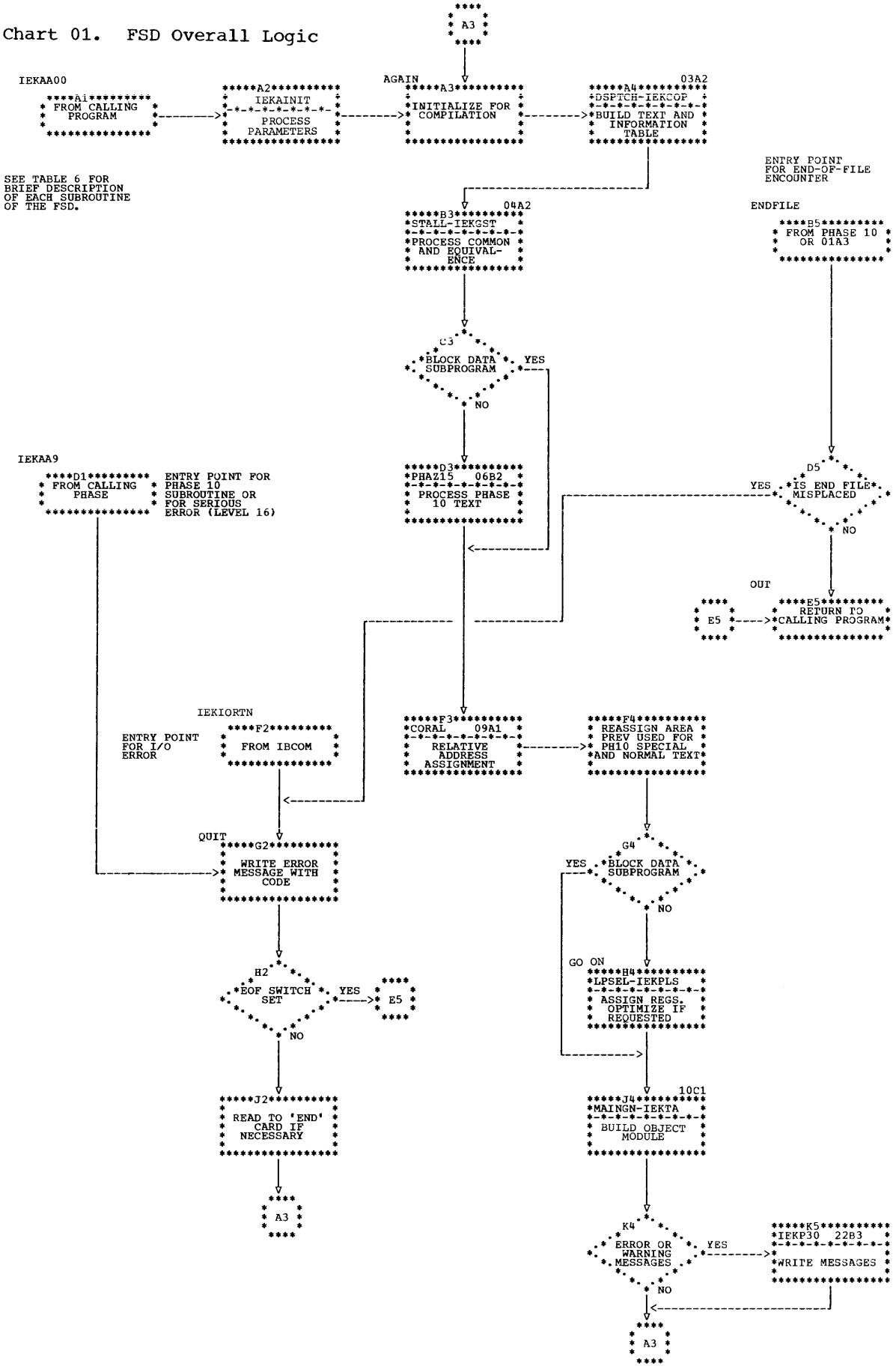


Chart 02. FSD Storage Distribution

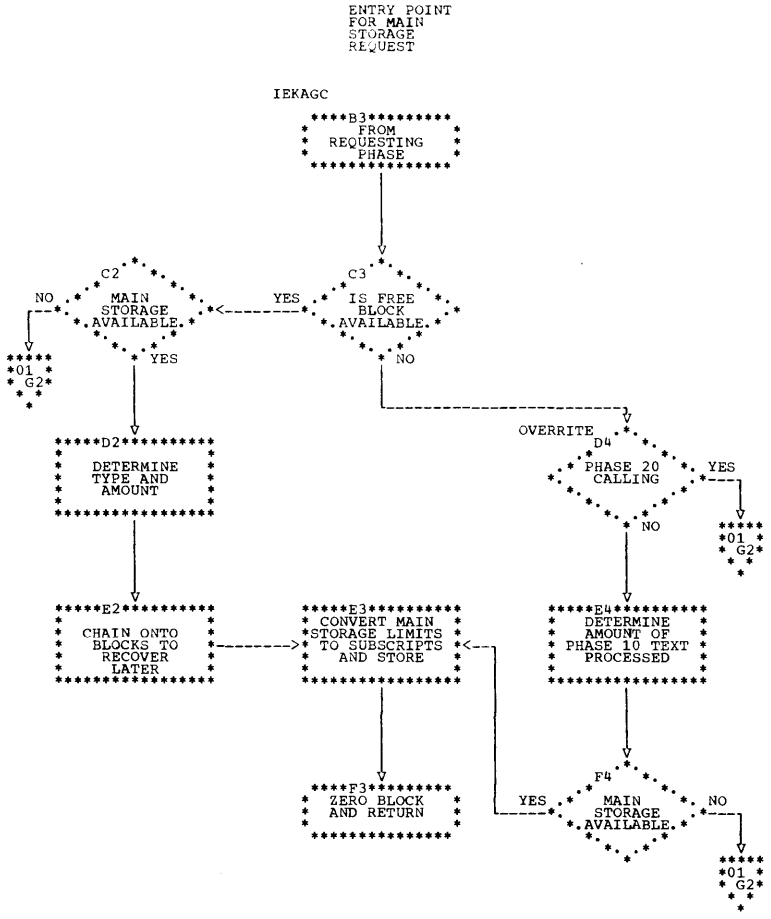


Table 6. FSD Subroutine Directory (Part 1 of 2)

Subroutine	Function
ADCON- IEKAAD	Internal adcon table.
AFIXPI- IEKAFP (AFIXPI)* (FIXPI)* (FIXPI#)*	Performs exponentiation of integers.
DCLIST- IEKTDC	Prints out assembly listing of source program.
ERCOM- IEKAER	Error message table.
IEKAAA	Communication table.
IEKAA00	Initializes compiler processing and calls the phases for execution. Entry point for compiler.
(ENDFILE)*	Receives control when end of data set is detected on input. Returns control to operating system.
(IEKAA9)*	IEKAA9 deletes compilation if requested.
(IEKAGC)*	IEKAGC allocates and keeps track of main storage used in the construction of the information table and for collecting text entries.
(IEKIORTN)*	Entry from IBCOM on I/O error.
IEKAA01	Defines default options.
IEKAA02 (PAGEHEAD)*	Defines DDNAMES for the compiler and page headings. Common area for IEKAA00 and IEKAINIT.
IEKAINIT	Processes parameters for OS/360 and gets core for the compiler.
IEKATB	Provides diagnostic dumps of internal text and tables.
IEKATM (PHASB)* (PHASS)* (PHAZSS)* (TIMERC)* (TOUT)* (TSP)* (TST)*	Timing routine.
IEKFCOMH (IBCOM)* (IBCOM#)*	Controls formatted compile-time input/output. (Corresponds to Library routine IHCFCOMH.)
IEKFIOCS (FIOCS)* (FIOCS#)*	Interface between compiler, IEKFCOMH, and QSAM.
*Secondary entry point	

Table 6. FSD Subroutine Directory (Part 2 of 2)

Subroutine	Function
IEKTLOAD (ESD)* (IEKUND)* (IEKURL)* (IEKUSD)* (IEKTXT)* (IEND)* (RLD)* (TXT)*	Builds ESD, TXT, RLD, and loader END records.
PUTOUT- IEKAPT (PUTOUT)*	Maximizing service routine for integers and reals, diagnostic trace routine; bypasses IEKFCOMH for some error messages.
*Secondary entry point	

Chart 03. Phase 10 Overall Logic

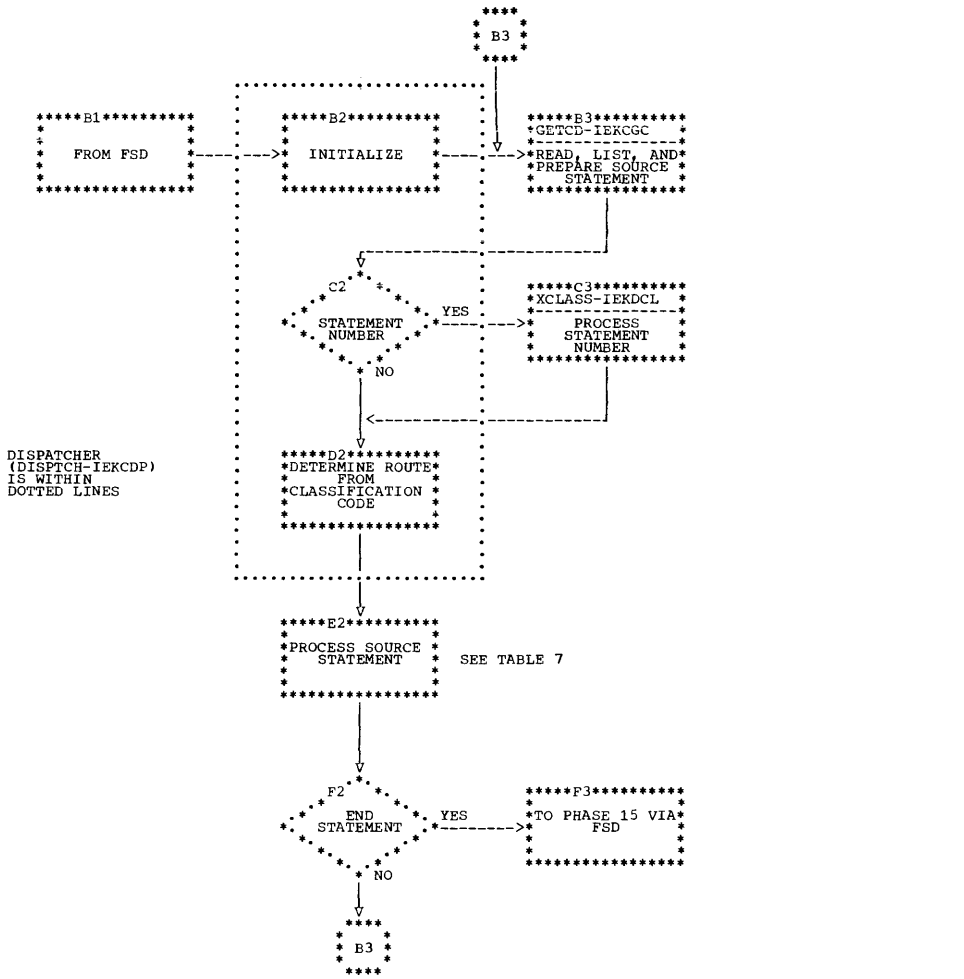


Chart 04. Subroutine STALL-IEKGST

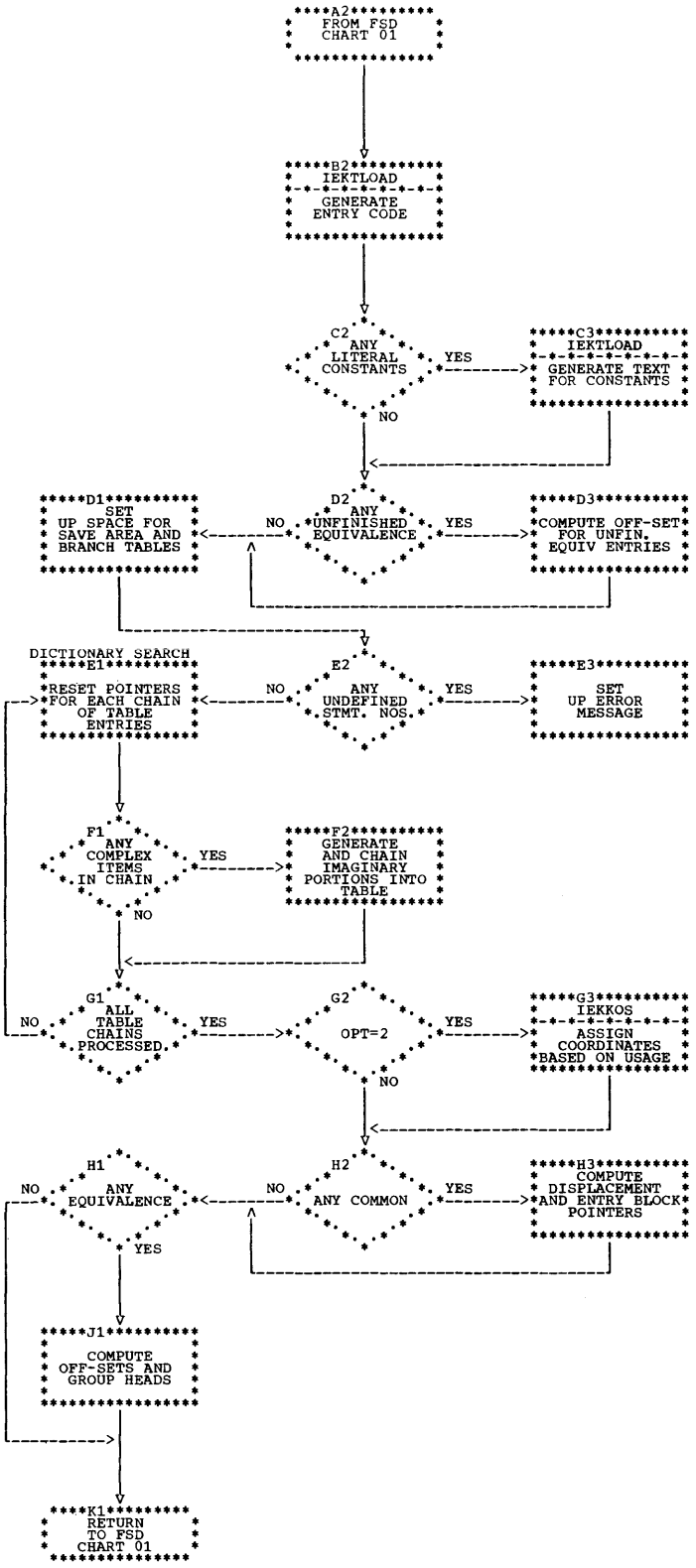


Table 7. Phase 10 Source Statement Processing

Statement Type	Main Processing Subroutine	Subroutines Used
Arithmetic	XARITH-IEKCAR	IEKCCR, IEKCDP, IEKCGW, IEKCPX, IEKCS1, IEKCS2
Statement Function	DSPTCH-IEKCDP XARITH-IEKCAR	IEKCCR, IEKCDP, IEKCGW, IEKCPX, IEKCS1, IEKCS2
DIMENSION, EQUIVALENCE, COMMON	XSPECS-IEKCSP	IEKCCR, IEKCDP, IEKCGW, IEKCLC, IEKCS1, IEKCS2, IEKCS3
EXTERNAL	DSPTCH-IEKCDP	IEKCGW, IEKCS3
Type, DATA	XDATA-IEKCDT	IEKCGW, IEKCLC, IEKCDP, IEKCCR, IEKCPX, IEKCS3, IEKCSP, IEKCS2
DO	XDO-IEKCDO	IEKCGW, IEKCDP, IEKCLT, IEKCS3, IEKCCR, IEKCS2, IEKCPX
SUBROUTINE, CALL, ENTRY, FUNCTION	XSUBPG-IEKCSR	IEKCGW, IEKCDP, IEKCS3, IEKCLC, IEKCLT IEKCPX
READ, WRITE, PRINT, PUNCH, FIND	XIOOP-IEKCI0	IEKCAR, IEKCCS, IEKCDP, IEKCGW, IEKCLT, IEKCPX, IEKCS1, IEKCS2, IEKCS3
DEFINE FILE, IMPLICIT, STRUCTURE, NAMELIST	XTNDED-IEKCTN	IEKCGW, IEKCDP, IEKCCR, IEKCS1, IEKCLC, IEKCS2, IEKCPX, IEKCS3
BACKSPACE, REWIND, END FILE, RETURN, ASSIGN, FORMAT, PAUSE, STOP, END	XIOPST-IEKDIO	IEKCGW, IEKCDP, IEKCPX, IEKCCR, IEKCLT, IEKCS2, IEKCS3
IF, CONTINUE, BLOCK DATA	DSPTCH-IEKCDP	IEKCPX
GO TO	XGO-IEKCGO	IEKCDP, IEKCGW, IEKCLT, IEKCPX, IEKCS3

Table 8. Phase 10 Subroutine Directory (Part 1 of 3)

Subroutine	Type	Function
CSORN-IEKCCR (IEKCLC)* (IEKCS1)* (IEKCS2)* (IEKCS3)*	Utility (collection, conversion, entry placement)	Secondary entry point IEKCCR directs the entering of variables and constants into information table Secondary entry point IEKCLC converts integer, real, and complex constants to their binary equivalents. Secondary entry point IEKCS1 places variable names on full word boundaries for comparison to other variable names. Secondary entry point IEKCS2 places dictionary entries constructed for variables and constants of the source module into the information table. Secondary entry point IEKCS3 combines the functions of entries IEKCS1 and IEKCS2 (above) for variable names.
DSPTCH-IEKCDP (IEKCIN)*	Dispatcher, Keyword, and Utility (entry placement)	Controls phase 10 processing, passes control to the preparatory subroutine to prepare the source statement, determines from the code assigned to the statement which subroutine is to continue processing the statement, and passes control to that subroutine. Develops intermediate text representations of the BLOCK DATA, CONTINUE, EXTERNAL, and IF statements and that portion of a statement function to the left of the equal sign; builds information table entries for the operands of these statements; and analyzes these statements for syntactical errors. Builds error table entries for the syntactical errors detected by phase 10 and places them in the error table. IEKCIN is the initial entry point to IEKCDP.
FORMAT-IEKTFM	Miscellaneous	Generates format text from phase 10 intermediate text.
GETCD-IEKCGC (IEKAREAD)*	Preparatory	Reads, lists (if requested), packs, and classifies each source statement. IEKAREAD is a secondary entry point to IEKCGC.
GETWD-IEKCGW	Utility (collection)	Obtains the next group of characters in the source statement being processed.
IEKKOS	Utility (table entry)	Assigns coordinates based on usage count to variables and constants.

*Secondary entry point

Table 8. Phase 10 Subroutine Directory (Part 2 of 3)

Subroutine	Type	Function
IEKXRS	Miscellaneous	Writes XREF buffer on SYSUT2.
LABTLU-IEKCLT	Utility (entry placement)	Places statement number entries into the information table.
PH10-IEKCAA	Utility (common data area)	Work area and communication region for phase 10.
PUTX-IEKCPX	Utility (entry placement)	Places text entries into the appropriate subblocks, obtains the next operator from the source statement, and places the operator in the text entry work area.
STALL-IEKGST	Utility (table entry and text generation)	Generates entry code for object module, calls IEKTFM to translate format text to object code, generates object code for literal data text, processes equivalence entries (those that were equivalenced before being dimensioned), sets aside space in the object module for the problem program save area and for computed GO TO branch tables, checks for undefined statement numbers, rechains variables, assigns coordinates based on usage count, processes COMMON entries, and processes EQUIVALENCE entries.
XARITH-IEKCAR	Arithmetic	Controls the processing of arithmetic statements, CALL arguments, expressions in IF statements, I/O list items, the expression portion of a statement function, and the branch tables of an arithmetic IF statement. Builds information table entries for the operands of the previously mentioned statements, and analyzes the statements for syntactical errors.
XCLASS-IEKDCL	Utility (text generation)	Controls the processing of source and compiler-generated statement numbers, generates the intermediate text required to increment a DO index and to compare the index with its maximum value, and processes CALL arguments of the form &label.
XDATYP-IEKCDT	Keyword (table entry and text generation)	Develops intermediate text representations of DATA and TYPE statements, information table entries for the operands of DATA and TYPE statements, and analyzes these statements for syntactical errors.
XDO-IEKCDO	Keyword (table entry and text generation)	Develops the intermediate text and information table entries for the DO statement and implied DOs appearing in input/output statements and analyzes them for syntactical errors.

Table 8. Phase 10 Subroutine Directory (Part 3 of 3)

Subroutine	Type	Function
XGO-IEKCGO	Keyword (table entry and text generation)	Develops intermediate text representations of the GO TO (unconditional, assigned, and computed) statements, constructs information table entries for the operands of these statements, and analyzes these statements for syntactical errors.
XIOOP-IEKCIO	Keyword (table entry and text generation)	Develops intermediate text representations of input/output statements, constructs information table entries for their operands, and analyzes input/output statements for syntactical errors. (I/O list items are processed by subroutine XARITH-IEKCAR.)
XREF-IEKXRF	Miscellaneous	Reads in XREF buffer from SYSUT2. Prints out a cross-reference listing directly after the source listing.
XSPECS-IEKCSP	Keyword (table entry)	Constructs information table entries for variables and arrays appearing in COMMON, DIMENSION, and EQUIVALENCE statements and analyzes these statements for syntactical errors.
XSUBPG-IEKCSR	Keyword (table entry and text generation)	Develops intermediate text representations of CALL, SUBROUTINE, ENTRY, and FUNCTION statements; constructs information table entries for the operands of these statements; and analyzes these statements for syntactical errors. (This subroutine passes control to subroutine XARITH-IEKCAR to process the arguments appearing in CALL statements.)
XTNDED-IEKCTN	Keyword (table entry and text generation)	Develops intermediate text for NAMELIST and DEFINE FILE statements; constructs information table entries for variables and arrays appearing in the NAMELIST, DEFINE FILE, and STRUCTURE statements; resets the implicit mode table according to the specification of the IMPLICIT statement; and analyzes these statements for syntactical errors.
XIOPST-IEKDIO	Keyword (table entry and text generation)	Develops intermediate text representations of ASSIGN, RETURN, FORMAT, PAUSE, BACKSPACE, REWIND, END FILE, STOP, and END statements; constructs information table entries for the operands of the ASSIGN, BACKSPACE, REWIND, and END FILE statements; and for the operands (if any) of the RETURN, PAUSE, and STOP statements; and analyzes all of these statements for syntactical errors.

Chart 05. Phase 15 Overall Logic

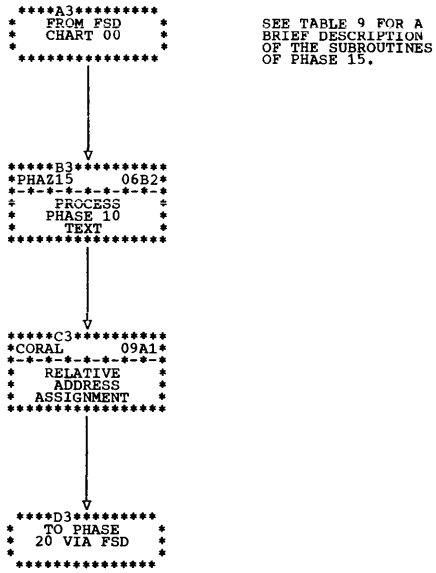


Chart 06. PHAZ15 Overall Logic

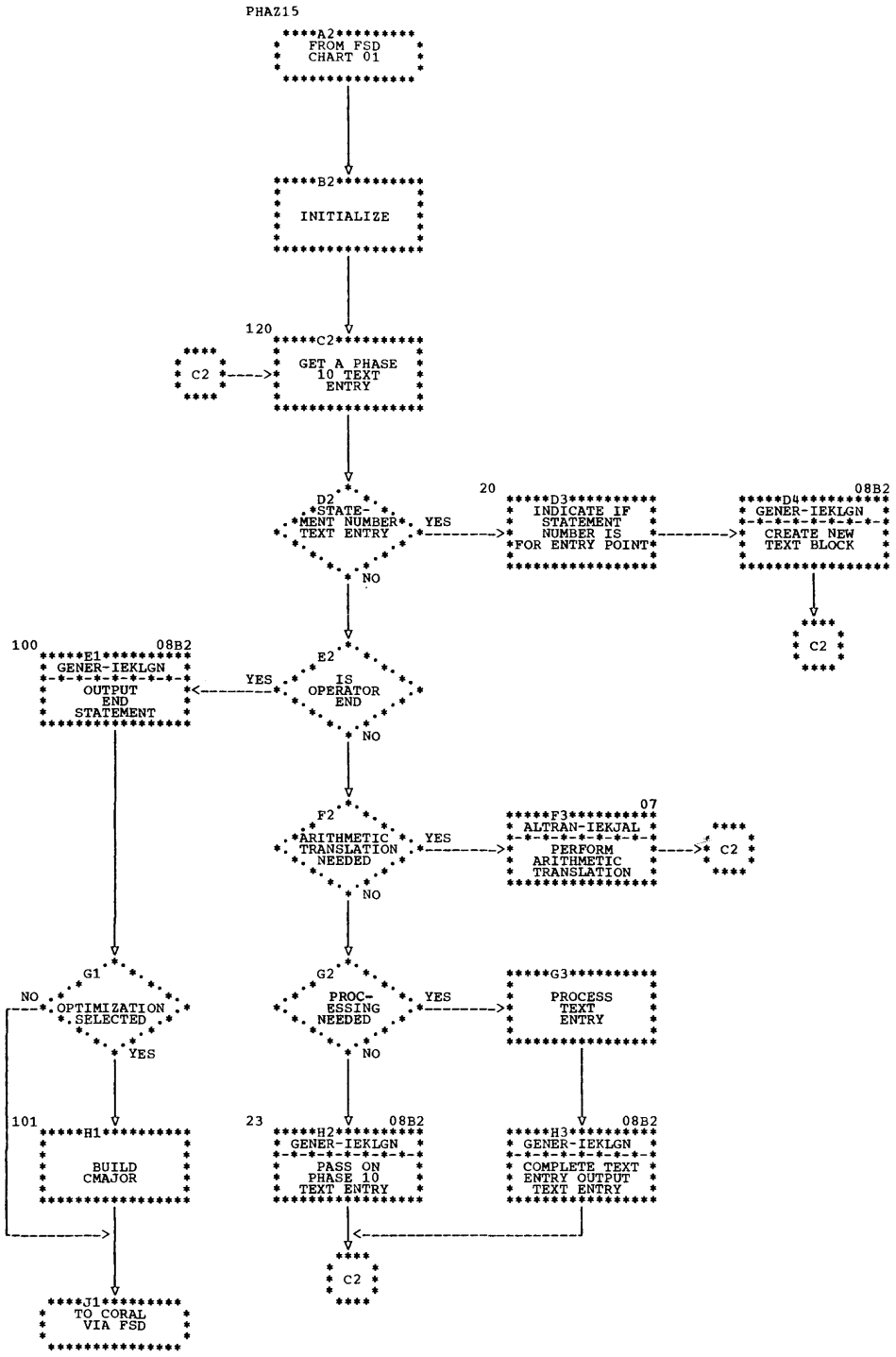
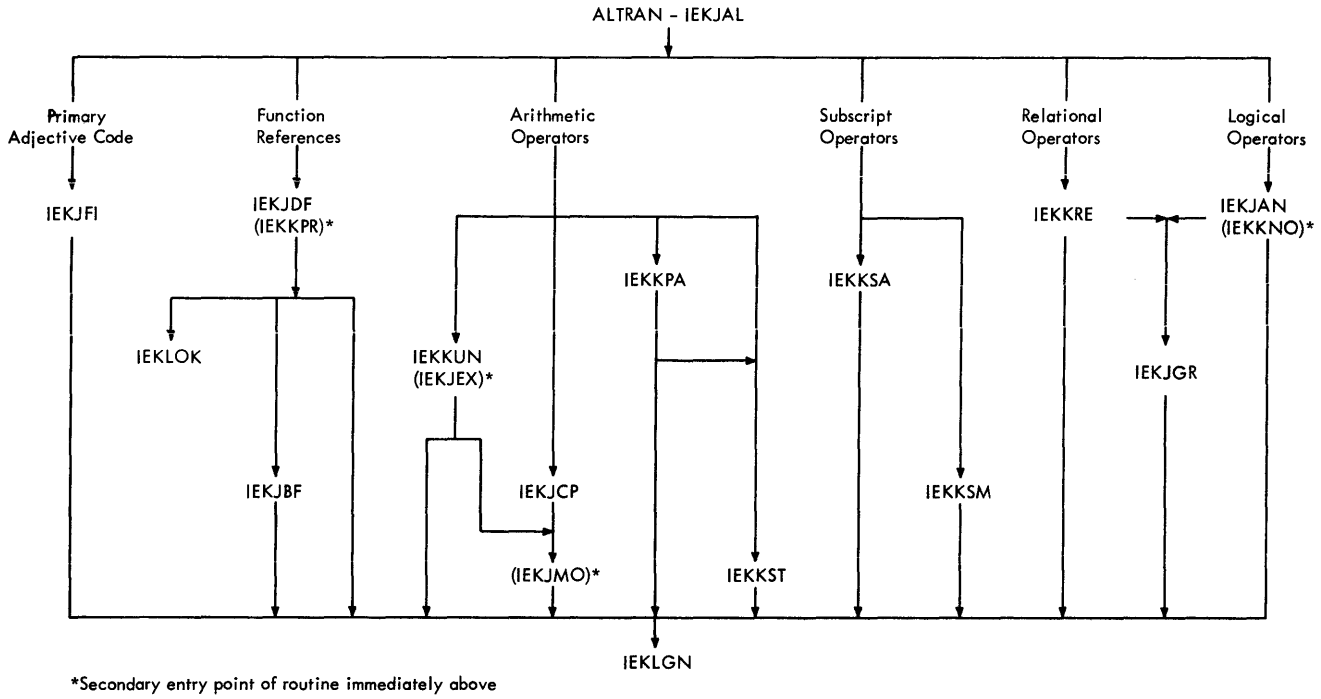


Chart 07. ALTRAN-IEKJAL Control Flow



*Secondary entry point of routine immediately above

NOTE: The logic and flow of the arithmetic translator is too complex to be represented on one or two conventional flowcharts. Chart 07 indicates the relationship between the arithmetic translator (subroutine ALTRAN) and its lower-level subroutines. An arrow flowing between two subroutines indicates that the subroutine at the origin of the arrow may, in the course of its processing, call the subroutine indicated by the arrowhead. In some cases, a subroutine called by ALTRAN may, in turn, call one or more subroutines to assist in the performance of its function. The level and sequence of subroutines is indicated by the lines and arrowheads.

In reality, all of the pathways shown connecting subroutines are two-way; however, to simplify the chart, only forward flow has been indicated by the arrowheads. All of the subroutines return control to the subroutine that called them when they complete their processing. (If a subroutine detects an error serious enough to warrant the deletion of the compilation, the subroutine passes control to the FSD, rather than return control to the subroutine that called it.)

The specific functions of each of the subroutines associated with the arithmetic translator are given in the subroutine directory following the charts for phase 15.

Chart 08. GENER-IEKLGN Text Generation

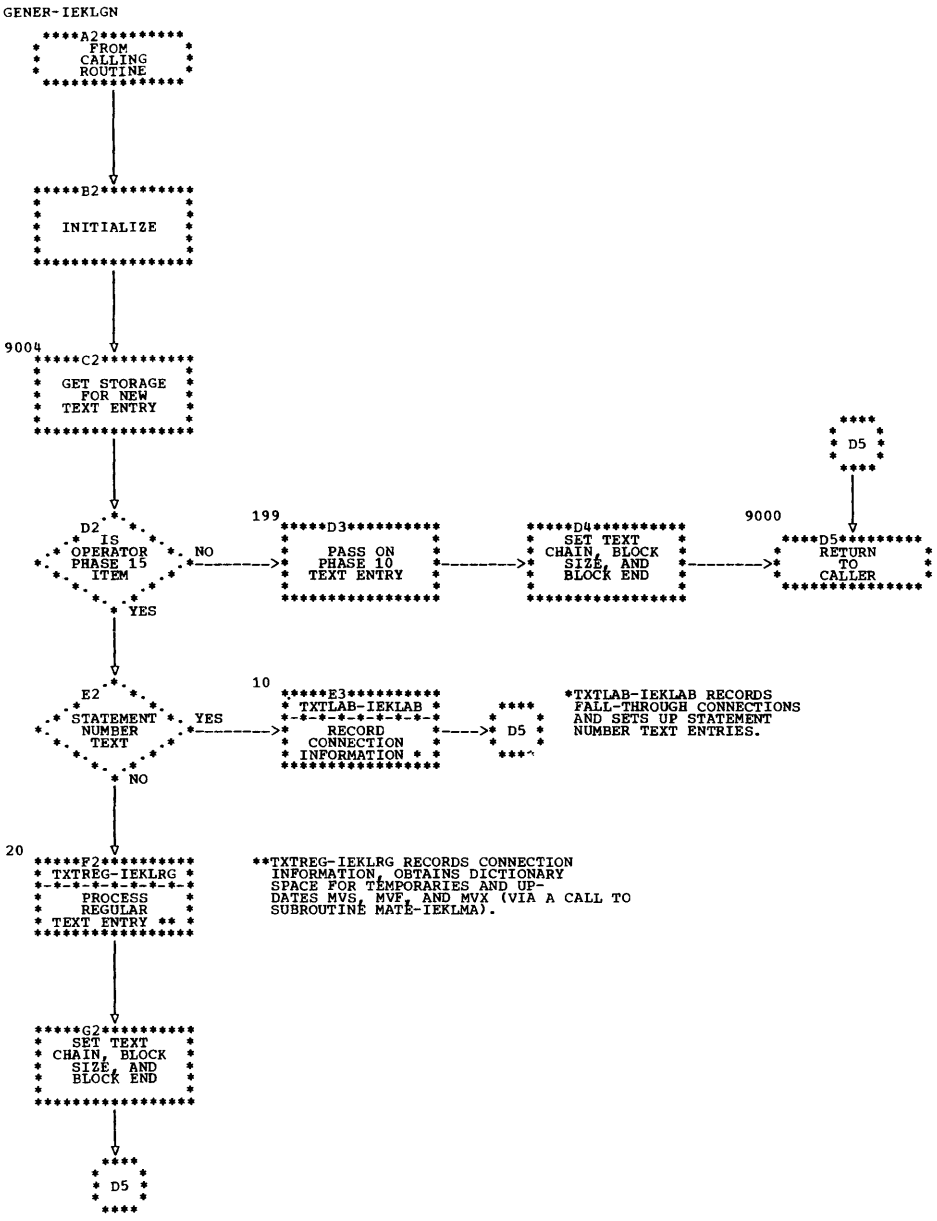


Chart 09. CORAL Overall Logic

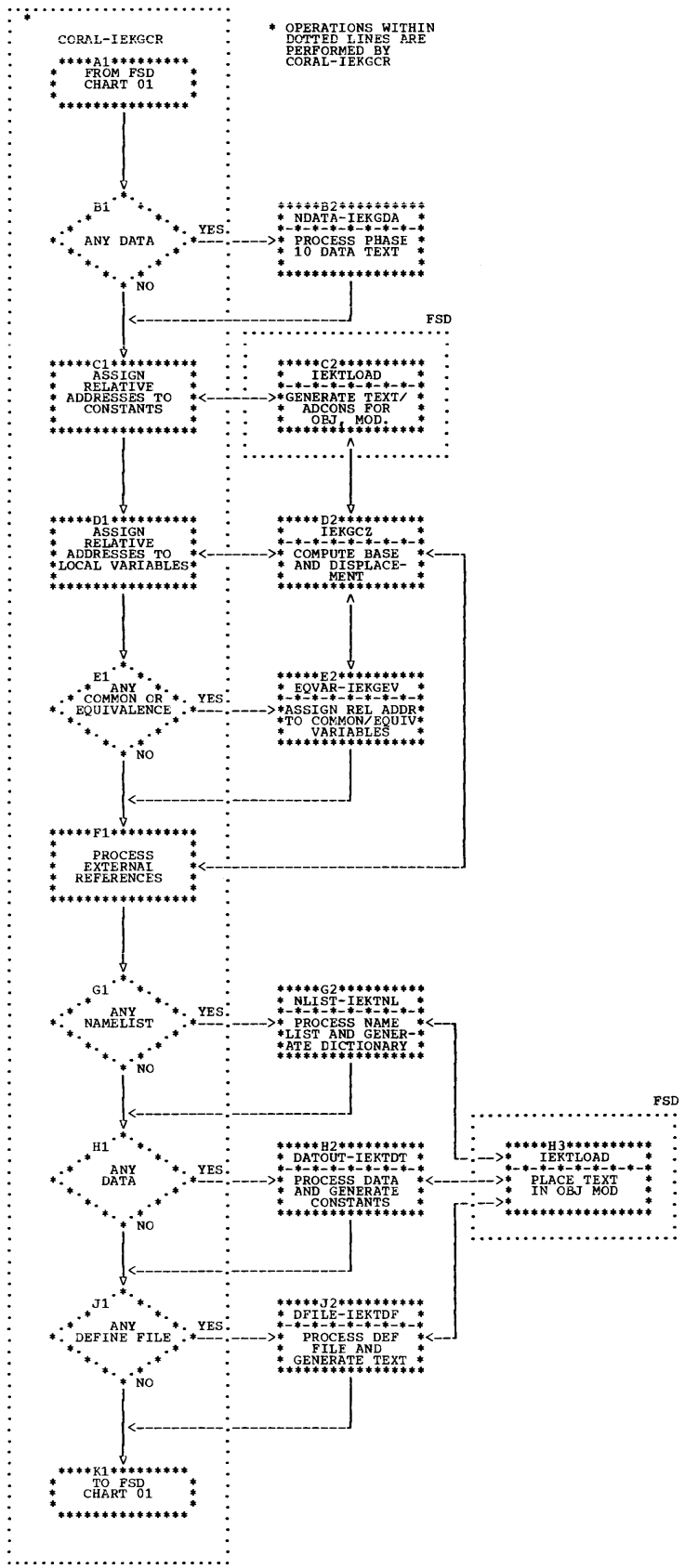


Table 9. Phase 15 Subroutine Directory (Part 1 of 2)

Subroutine	Associated Phase 15 Segment	Function
ALTRAN-IEKJAL	PHAZ15 (5)	Controls the arithmetic translation process.
ANDOR-IEKJAN (IEKKNO)*	PHAZ15 (5)	Checks the mode of the arguments passed to it, decomposes IF statements, and generates text entries for AND and OR operations.
BLTNFN-IEKJBF	PHAZ15 (5)	Generates phase 15 text for in-line functions by either expanding the function or creating a phase 15 text item (which is expanded by phase 25).
CNSTCV-IEKKCN	PHAZ15 (5)	Performs compile time conversion of constants.
CORAL-IEKGCR	CORAL (6)	Controls the flow of space allocation for variables, constants, and adcons necessary for local variables, COMMON, EQUIVALENCE, and external references; processes constants, local variables, and external references.
CMSIZE-IEKGCZ	CORAL (6)	Keeps track of space being allocated; generates adcons for address computation; rechains data text, generates adcons for COMMON, EQUIVALENCE, and external references; and sets up error table entries for phase 30.
CPLTST-IEKJCP (IEKJMO)*	PHAZ15 (5)	Checks the mode of the operands in an arithmetic triplet making adjustments where necessary and controls text generation for the triplet.
DATOUT-IEKTDT	CORAL (6)	Puts phase 15 data text into object module.
DFILE-IEKTDF	CORAL (6)	Processes define file text.
DFUNCT-IEKJDF (IEKKPR)*	PHAZ15 (5)	Determines if a reference is to an in-line, library, or external function, and determines the validity of arguments to the subprogram; inserts the appropriate function operator into phase 15 text and builds the parameter list in the adcon table and in text for the subprogram referred to; performs parameter list optimization.
DUMP15-IEKLER	PHAZ15 (5)	Records errors detected during PHAZ15 processing.
EQVAR-IEKGEV	CORAL (6)	Handles COMMON and EQUIVALENCE space allocation.
FINISH-IEKJFI	PHAZ15 (5)	Completes the processing required for a statement when its primary adjective code is forced from the pushdown table.
FUNRDY-IEKJFU	PHAZ15 (5)	Creates pushdown entries for references to implicit library functions.
GENER-IEKLGN	PHAZ15 (5)	Generates phase 15 text consisting of unchanged phase 10 text, phase 15 standard text, and phase 15 statement number text.
GENRTN-IEKJGR	PHAZ15 (5)	Builds appropriate phase 15 text entries for simple items forced from the pushdown table.

*Secondary entry point

Table 9. Phase 15 Subroutine Directory (Part 2 of 2)

Subroutine	Associated Phase 15 Segment	Function
LOOKER-IEKLOK	PHAZ15 (5)	Searches the function table (IEKLTB) to determine if a given function is FORTRAN supplied.
MATE-IEKLMA	PHAZ15 (5)	Records usage information in the MVS, MVF, and MVX fields if one of the optimizer paths through phase 20 is selected.
NDATA-IEKGDA	CORAL (6)	Converts phase 10 data text to phase 15 data text.
NLIST-IEKTNL	CORAL (6)	Processes namelist text.
OP1CHK-IEKKOP (IEKKNG)*	PHAZ15 (5)	Determines whether or not operand 1 should be a temporary and checks for negative arguments.
PAREN-IEKKPA	PHAZ15 (5)	Removes the (or -(from the pushdown table when the corresponding) is encountered.
PHAZ15-IEKJA	PHAZ15 (5)	Controlling routine of PHAZ15. Determines if the phase 10 text for a statement needs arithmetic translation. If so, ALTRAN-IEKJAL is called. Otherwise GENER-IEKLGN is called to put out unchanged phase 10 text. Builds CMAJOR if OPT=2.
RELOPS-IEKKRE	PHAZ15 (5)	Calls subroutine GENER-IEKLGN to generate text entries for relational operators. (Output may be either a relational or branch operation.)
STTEST-IEKKST	PHAZ15 (5)	Builds text for replacement statements [e.g., A=B, A=B(I), A(I)=B, A(I)=B(I)].
SUBADD-IEKKSA	PHAZ15 (5)	Generates text to add the terms in a subscript computation, determines if a subscript text entry in the pushdown table should be entered into phase 15 text, and calls subroutine GENER-IEKLGN to generate the text entry when appropriate.
SUBMLT-IEKKSM	PHAZ15 (5)	Generates the text to multiply the first term of a subscript computation by its associated length factor, or, in the case of variable dimension, to multiply the nth dimension by length.
TXTLAB-IEKLAB	PHAZ15 (5)	Processes statement number text entries for subroutine GENER-IEKLGN and creates entries in RMAJOR.
TXTREG-IEKLRG	PHAZ15 (5)	Processes standard phase 15 text entries for subroutine GENER-IEKLGN and makes RMAJOR entries.
UNARY-IEKKUN (IEKKSW)* (IEKJEX)*	PHAZ15 (5)	Optimizes arithmetic triplets and processes the exponentiation operator.

*Secondary entry points.

Table 10. Phase 15 COMMON Areas

Name	Function
IEKGA1	CORAL COMMON data area.
PH15-IEKJA1	Phase 15 COMMON data area.
CMAJOR-IEKJA2	Backward connection table.
IEKJA3	Function information tables.
RMAJOR-IEKJA4	Forward connection table.
IEKLTB	Function table COMMON area.

Chart 10. Phase 20 Overall Logic

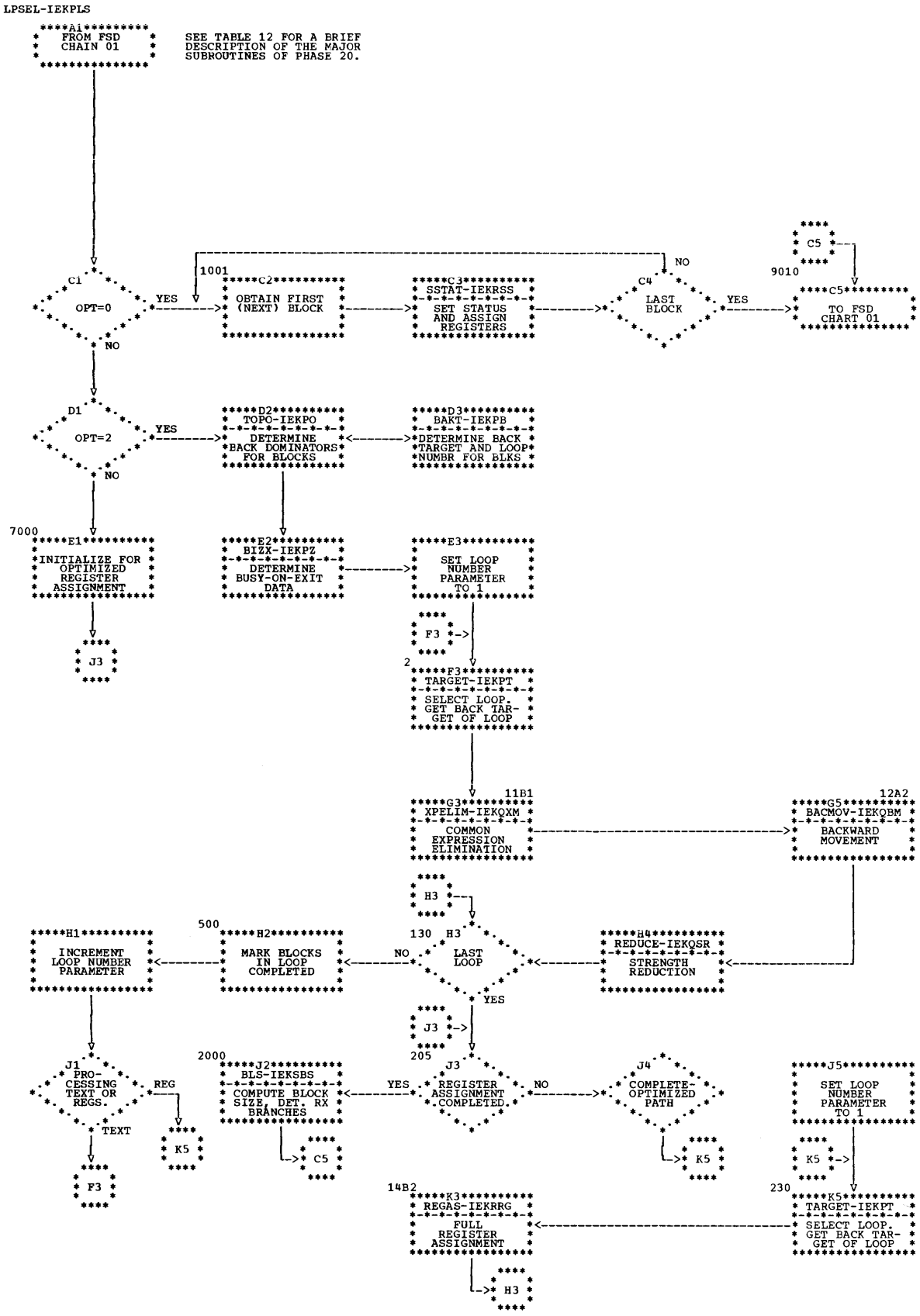


Chart 11. Common Expression Elimination (XPELIM-IEKQXM)

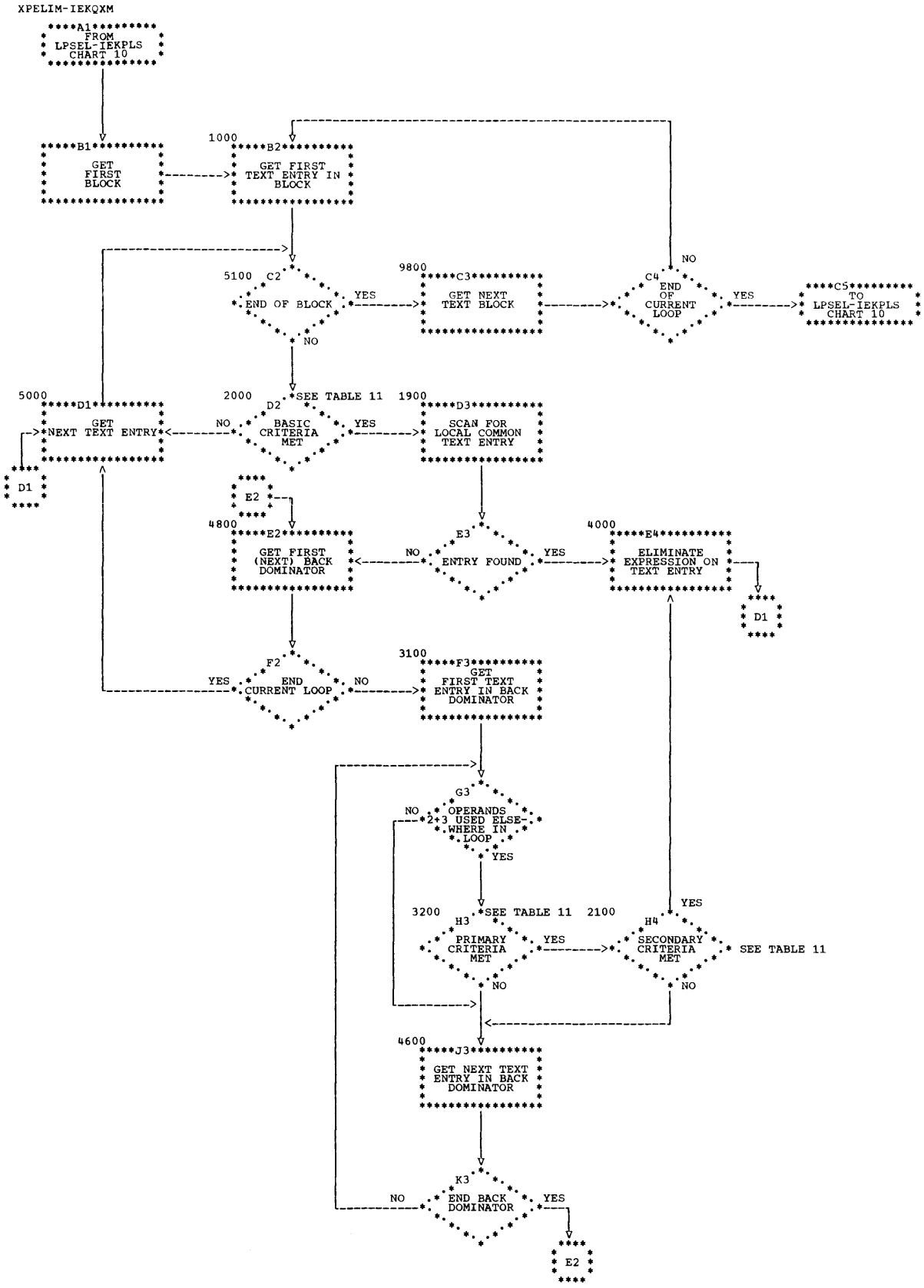


Chart 12. Backward Movement (BACMOV-IEKQBM)

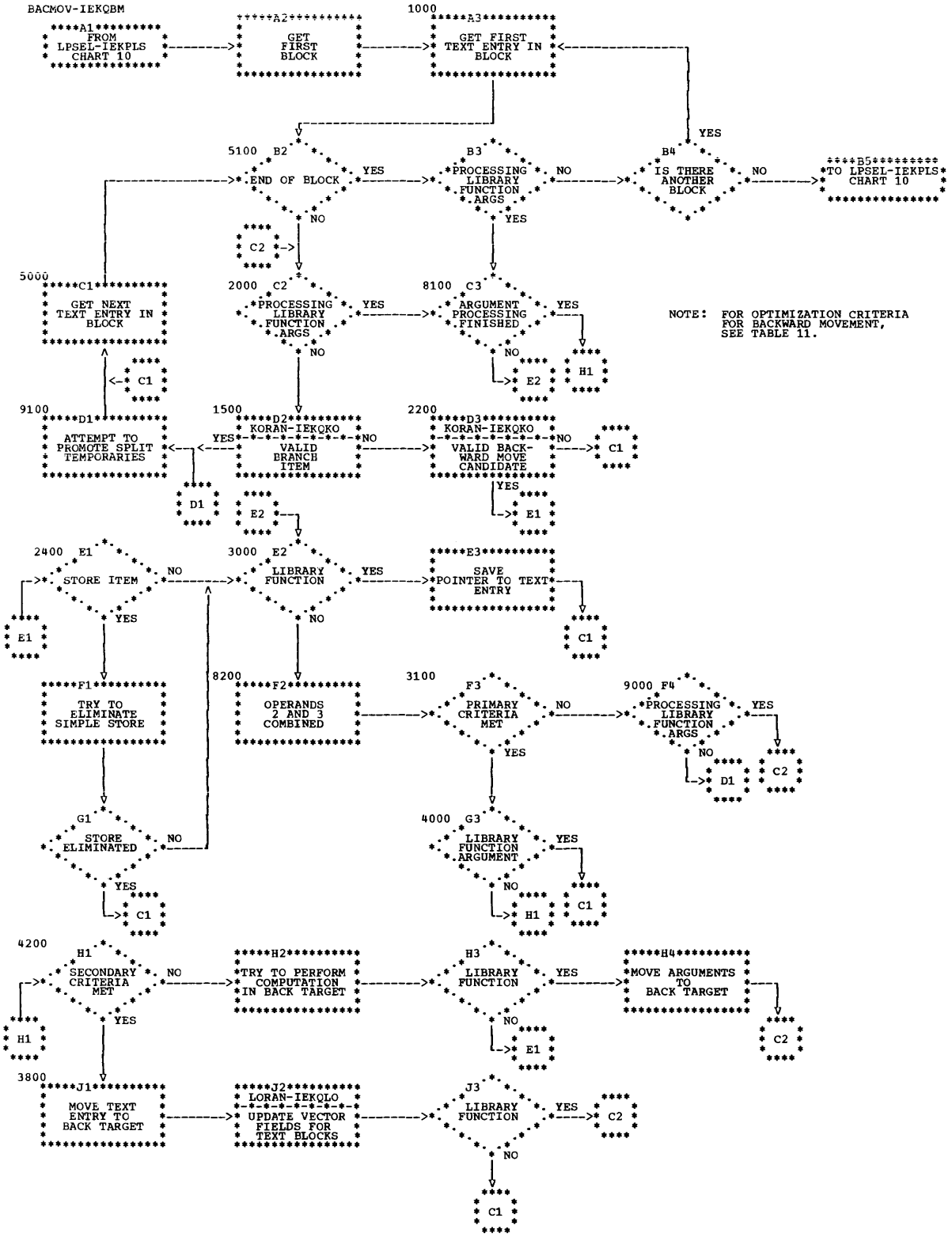


Chart 13. Strength Reduction (REDUCE-IEKQSR)

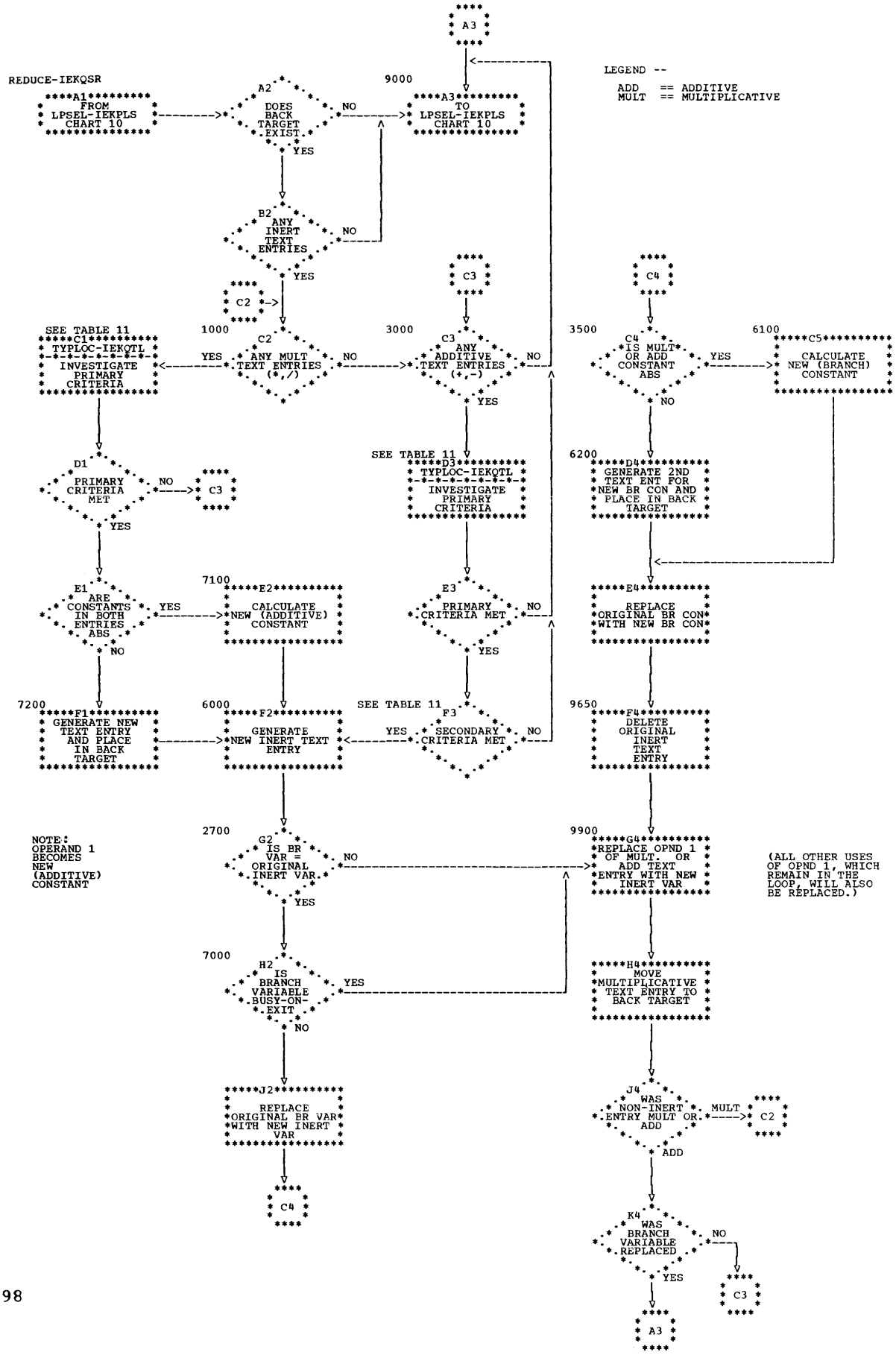


Chart 14. Full Register Assignment (REGAS-IEKRRG)

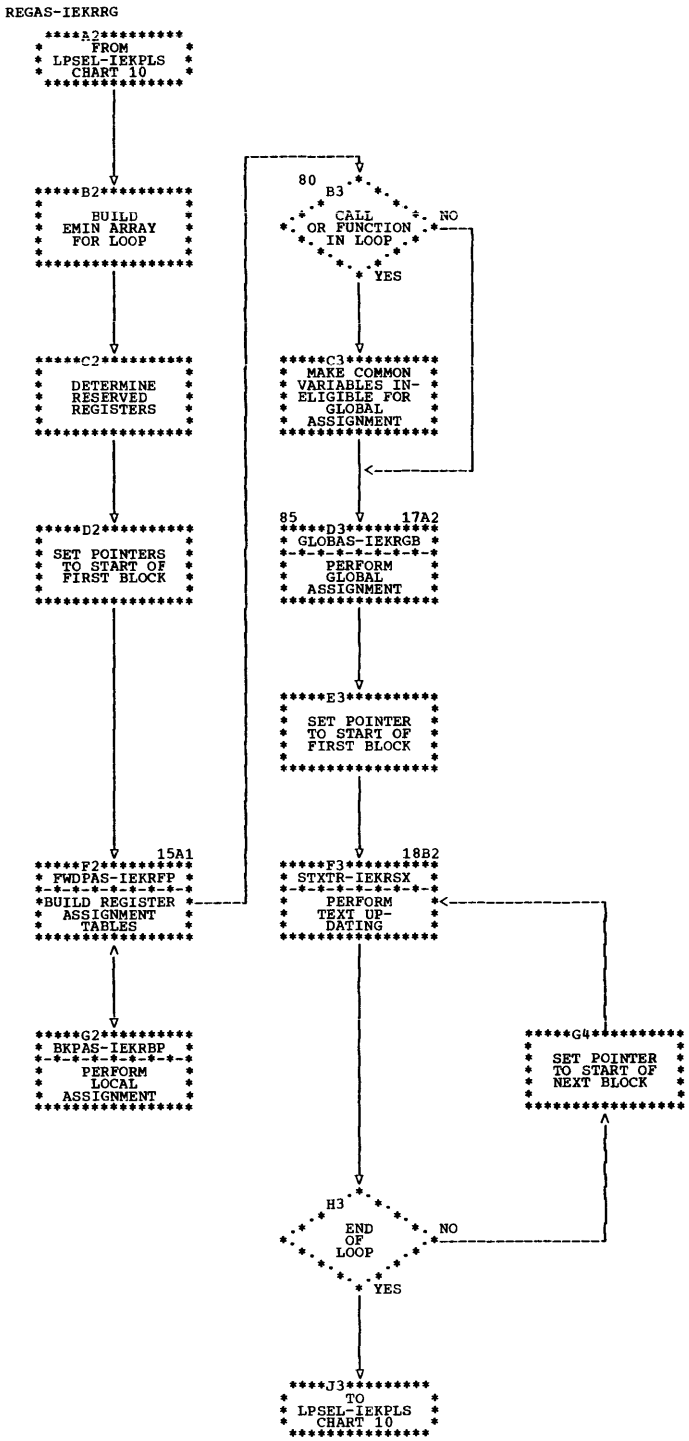


Chart 15. Table Building (FWDPAS-IEKRFP)

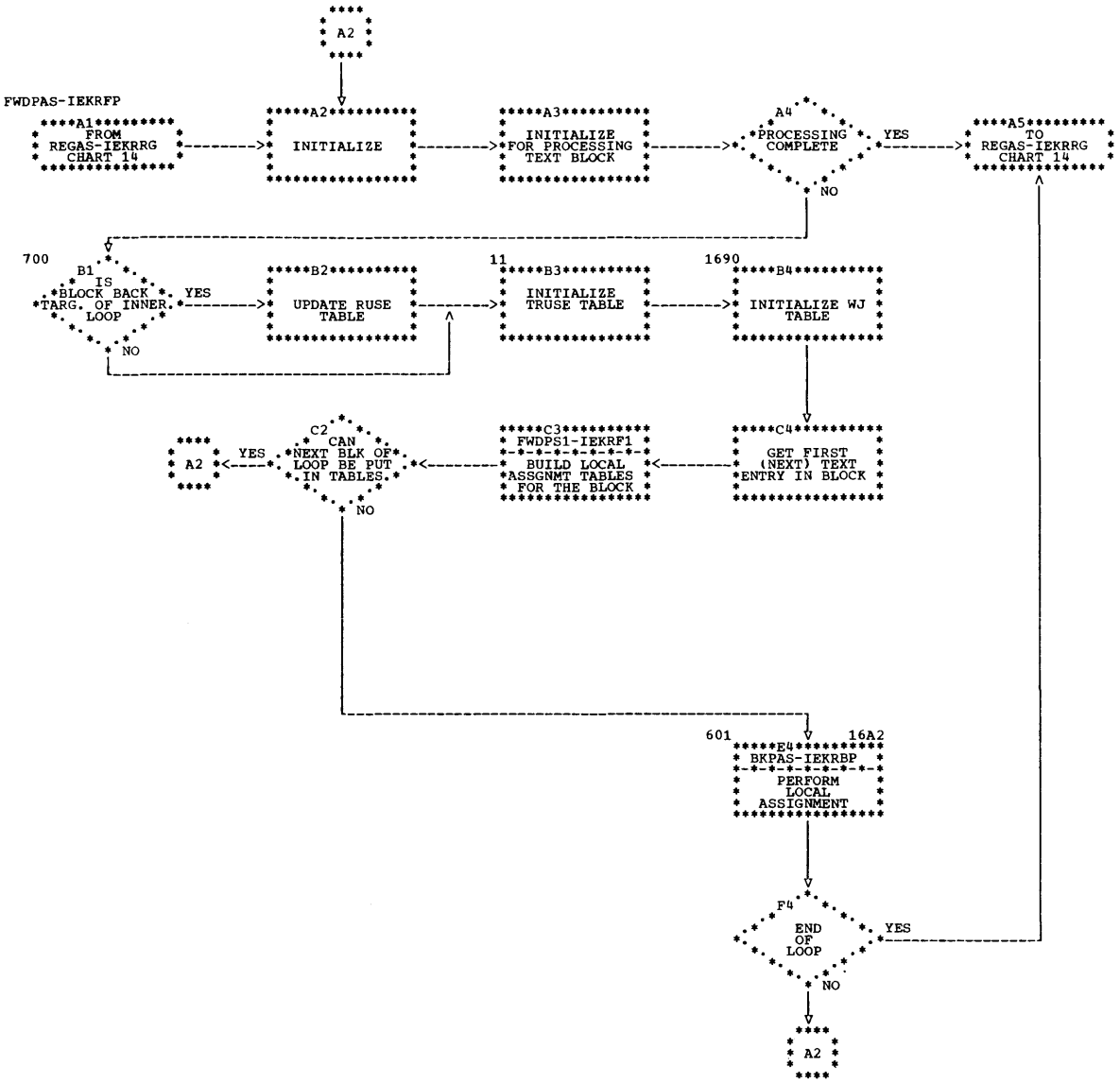


Chart 16. Local Assignment (BKPAS-IEKRBP)

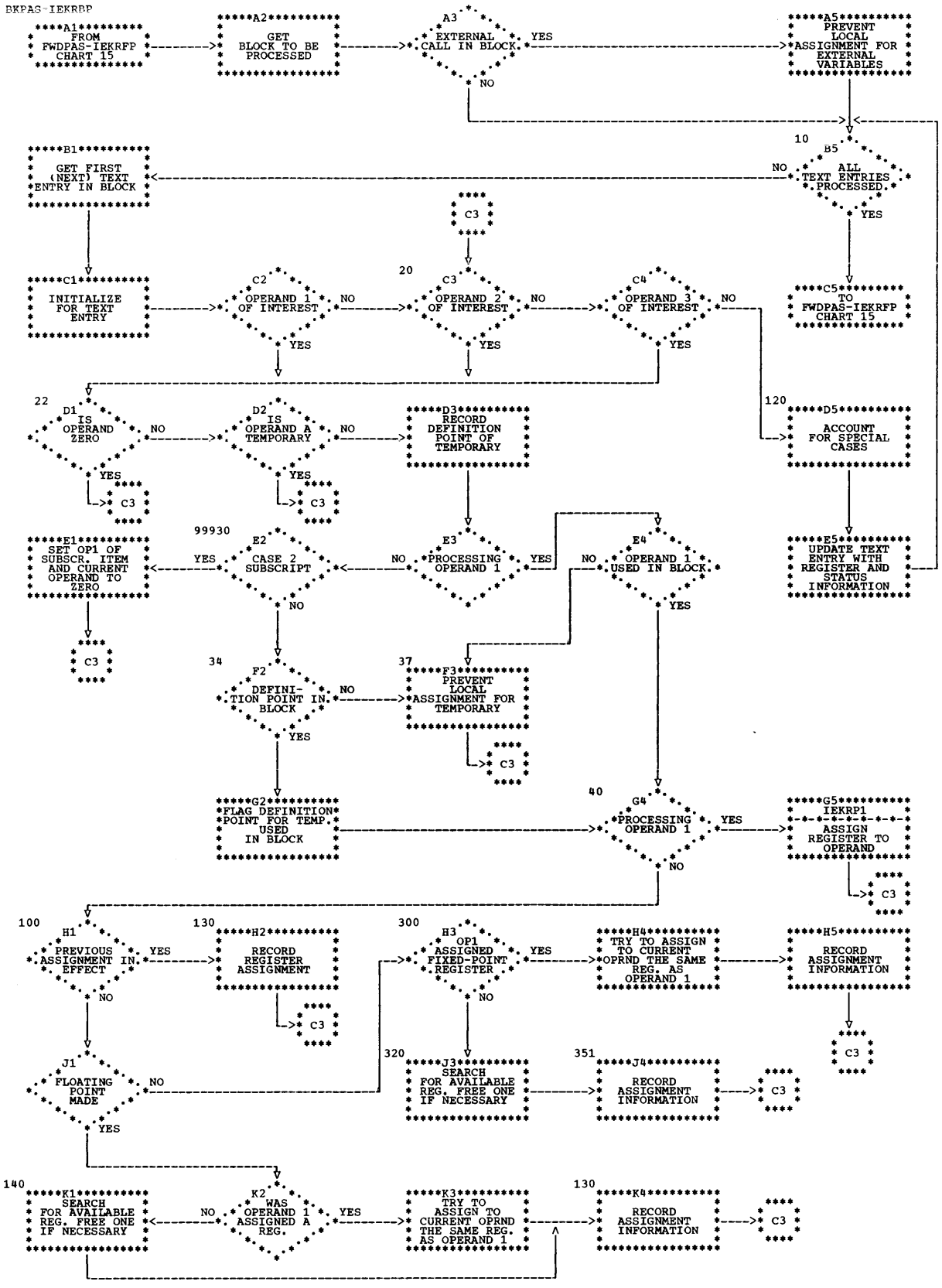


Chart 17. Global Assignment (GLOBAS-IEKRGB)

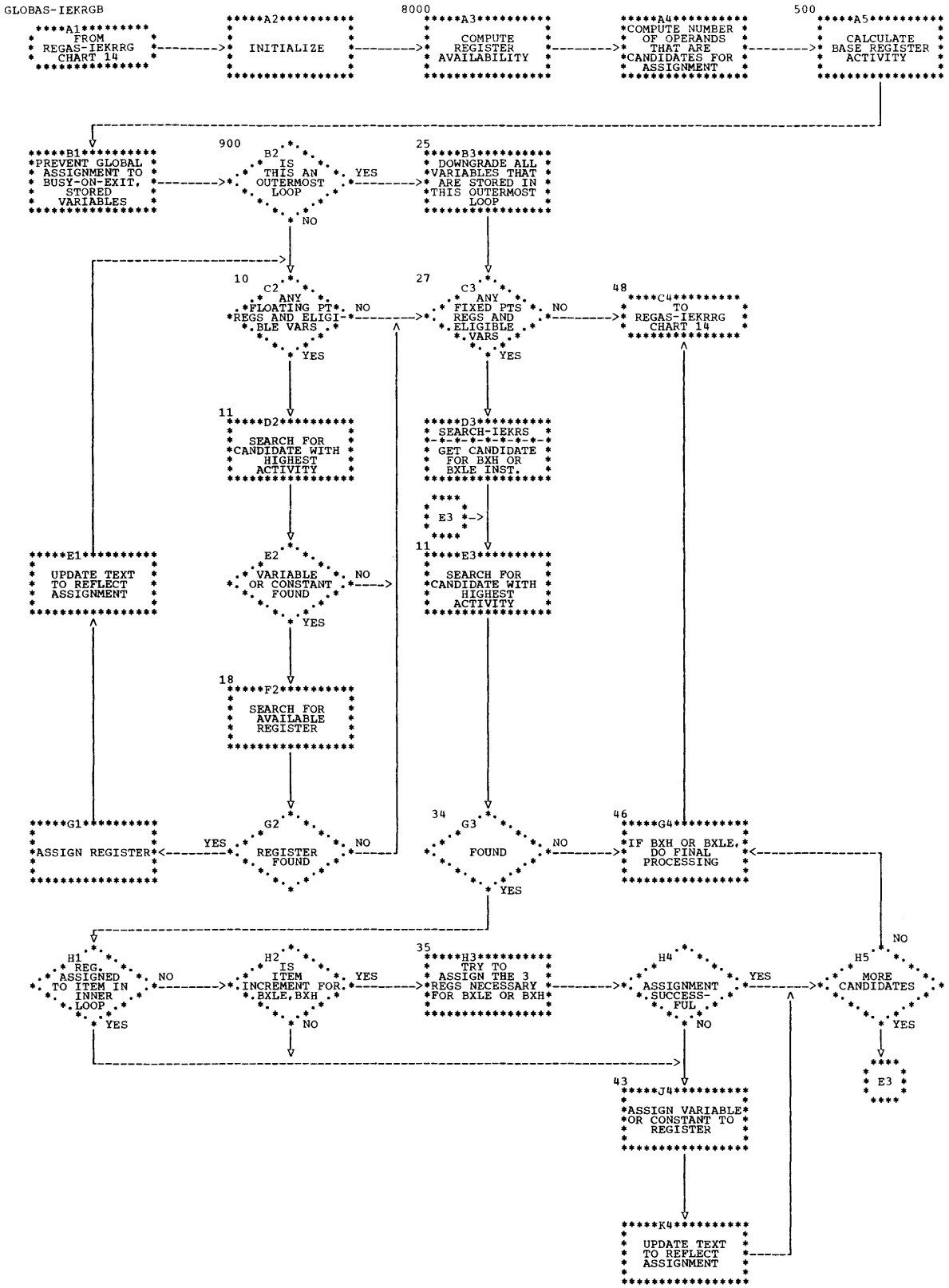


Chart 18. Text Updating (STXTR-IEKRSX)

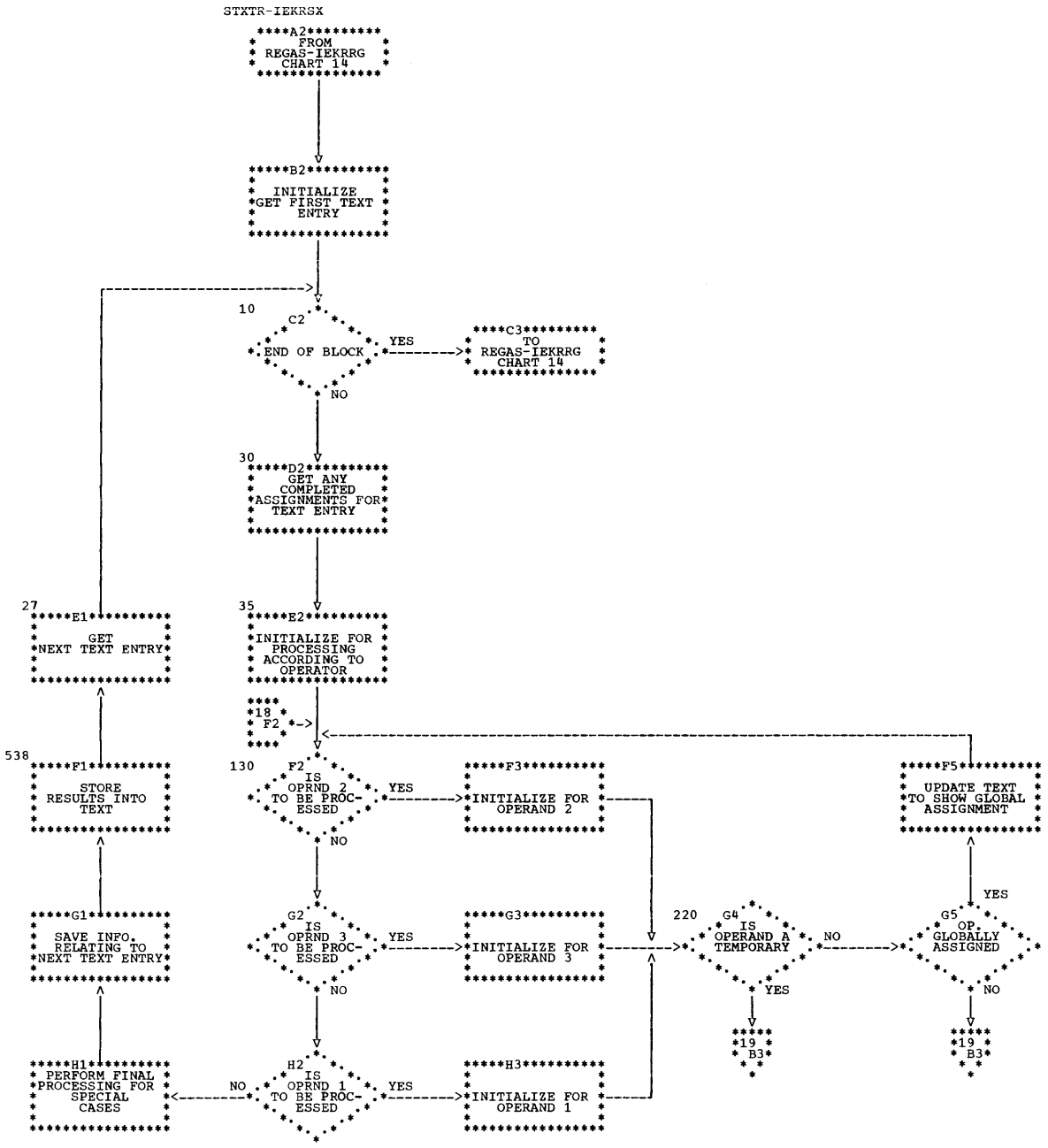


Chart 19. Text Updating (STXTR-IEKRSX) (Continued)

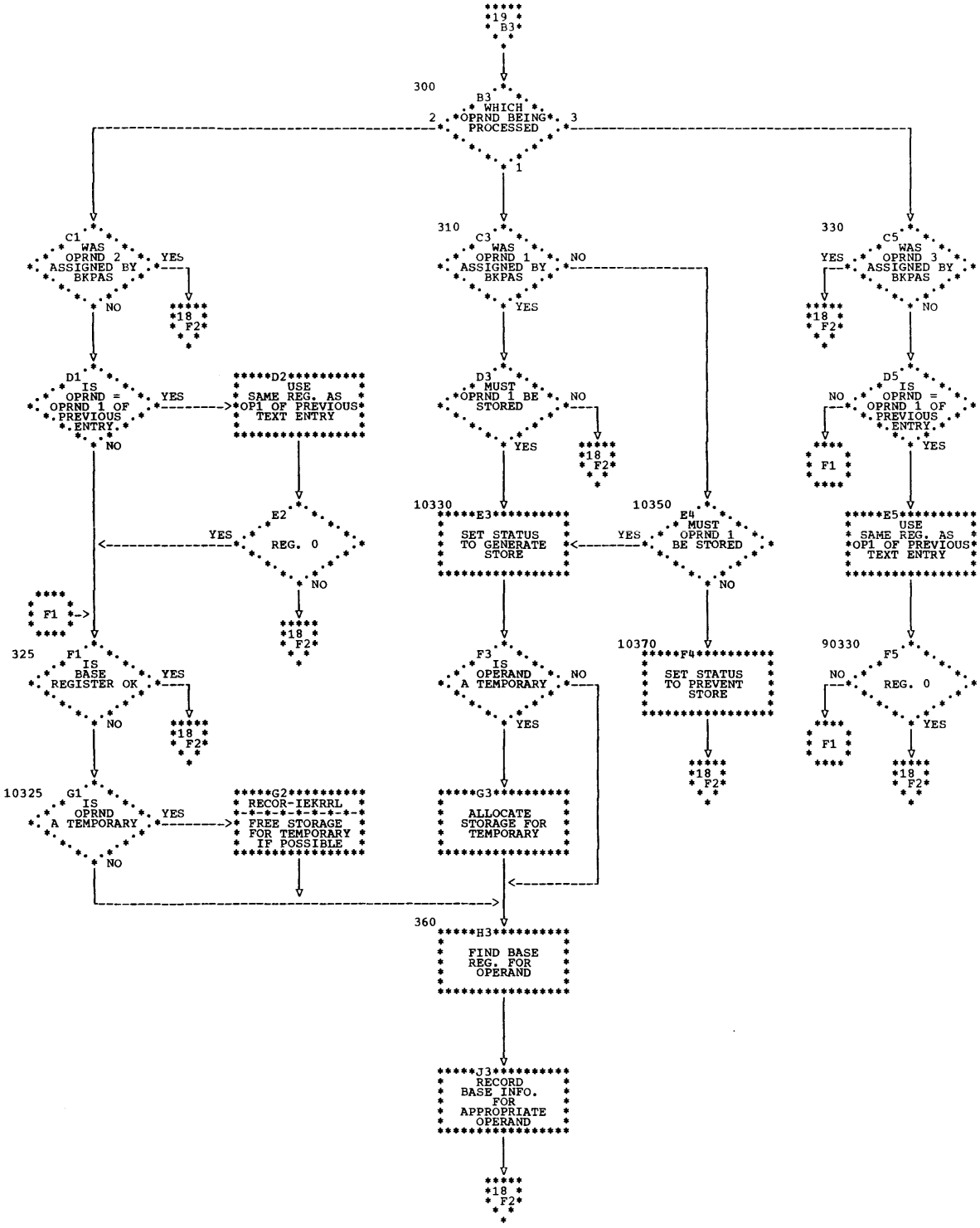


Table 11. Criteria for Text Optimization

Process	Basic	Primary	Secondary
Common Expression Elimination	Subscript, arithmetic, logical, or binary operator	Matching operand 2, operand 3, and operator	Matching operand 2, operand 3, and operator with no intervening redefinitions
Backward Movement	Arithmetic or logical operator	Operand 2 and operand 3 undefined in the loop	Operand 1 not busy on exit from target; operand 1 undefined elsewhere in the loop
Strength Reduction	Additive operator; inert variable	Interaction of inert variable with additive or multiplicative operator	Function of absolute constants or stored constants

Table 12. Phase 20 Subroutine Directory (Part 1 of 2)

Subroutine	Function	Type
BACMOV-IEKQBM	Controls backward movement, produces new inert text entries for strength reduction, builds type tables for strength reduction, and performs compile-time mode conversions.	Text optimization
BAKT-IEKPB	Computes the loop number of each module block.	Structural determination
BIZX-IEKPZ	Computes the proper MVX setting for each variable in each block of the module.	Structural determination
BKDMP-IEKRBK	Produces TRACE for full register assignment.	Register assignment
BKPAS-IEKRBP	Controls local register assignment.	Register assignment
BLS-IEKSBS	Computes the total size of each block in the module and determines which module blocks can be reached via RX-format branch instructions.	Branching optimization
CXIMAG-IEKRCI	Processes imaginary parts of complex functions during local register assignment.	Register assignment
FCLT50-IEKRFL (TNSFM-IEKRTF)* (RELCOR-IEKRRL)*	Performs special checks on text items whose function codes are less than 50.	Register assignment
	Secondary entry point TNSFM-IEKRTF performs special checks on text items whose function codes are in the range of 50 to 55 inclusive.	Register assignment
	Secondary entry point RELCOR-IEKRRL releases temporary main storage so it can be reused.	Register assignment
FREE-IEKRFR	Releases busy registers during overflow conditions (local assignment).	Register assignment
FWDPAS-IEKRFP	Table-building routine for full register assignment.	Register assignment
FWDPS1-IEKRF1	Determines whether or not text operands are register candidates prior to local register assignment.	Register assignment
GLOBAS-IEKRGB	Assigns most active variables to registers across the loop.	Register assignment
IEKPBL	BLOCK DATA subroutine for register assignment.	Register assignment
LOC-IEKRL1	COMMON data area for structural determination.	Structural determination
LPSEL-IEKPLS	Controls sequencing of loops and passes control to text optimization and register assignment routines	Control routine
REDUCE-IEKQSR	Controls strength reduction.	Text optimization

*Secondary entry point

Table 12. Phase 20 Subroutine Directory (Part 2 of 2)

Subroutine	Function	Type
REGAS-IEKRRG	Controls full register assignment.	Register assignment
SEARCH-IEKRS	Provides register loads upon entering the module.	Register assignment
SPLRA-IEKRSL	Assigns registers during basic register assignment.	Register assignment
SSTAT-IEKRSS	Sets status information for operands and base addresses of text entries.	Register optimization
STXTR-IEKRSX	Controls text updating.	Register assignment
TALL-IEKRLL	Assigns storage for temporaries.	Register assignment
TARGET-IEKPT	Identifies the members of a loop and its back target.	Text optimization
TOPO-IEKPO	Computes the immediate back dominator of each block in the module.	Structural determination
XPELIM-IEKQXM	Controls common expression elimination.	Text optimization
*Secondary entry point		

Table 13. Phase 20 Utility Subroutines

Subroutine	Function
CIRCLE-IEKQCL (FOLLOW-IEKQF)*	Examines composite vectors, or each local vector if necessary.
CLASIF-IEKQCF (PARFIX-IEKQPX)* (MODFIX-IEKQMF)*	Classifies operands of the current text entry, changes parameter list to correspond to text replacements, and adjusts text entry for possible mode change.
GETDIK-IEKPGK (FILTEX-IEKPFT)* (GETDIC-IEKPGC)* (INVERT-IEKPIV)* (OVFL-IEKPOV)*	Fills text space according to the arguments, gets space for temporaries, gets space for constants, and obtains previous text entry.
IEKARW	Calls FIOCS# to rewind the required data set.
IEKPOP	Common data area for phase 20.
KORAN-IEKQKO (LORAN-IEKQLO)*	Performs bit manipulation for text optimization, updates composite LMVS and LMVF matrixes.
MOVTEX-IEKQMT (DELTEX-IEKQDT)*	Moves text entries, deletes current text entry by rechaining, and updates MVS and MVF vectors.
PERFOR-IEKQPF	Performs combination of constants at compile time.
SRPRIZ-IEKQAA (-IEKQAB)*	Records structured source program listing on the SYSPRINT data set.
SUBSUM-IEKQSM	Replaces operands with equivalent values and, if possible, operand values with equivalent values.
TYPLOC-IEKQTL	Locates interaction of text entries for strength reduction.
WRITEX-IEKQWT	Prints diagnostic trace information when text optimization and TRACE option are specified.
XSCAN-IEKQXS (YSCAN-IEKQYS)* (ZSCAN-IEKQZS)*	Performs local block scan for backward movement, for common expression elimination, and for strength reduction.
*Secondary entry point	

Chart 20. Phase 25 Processing

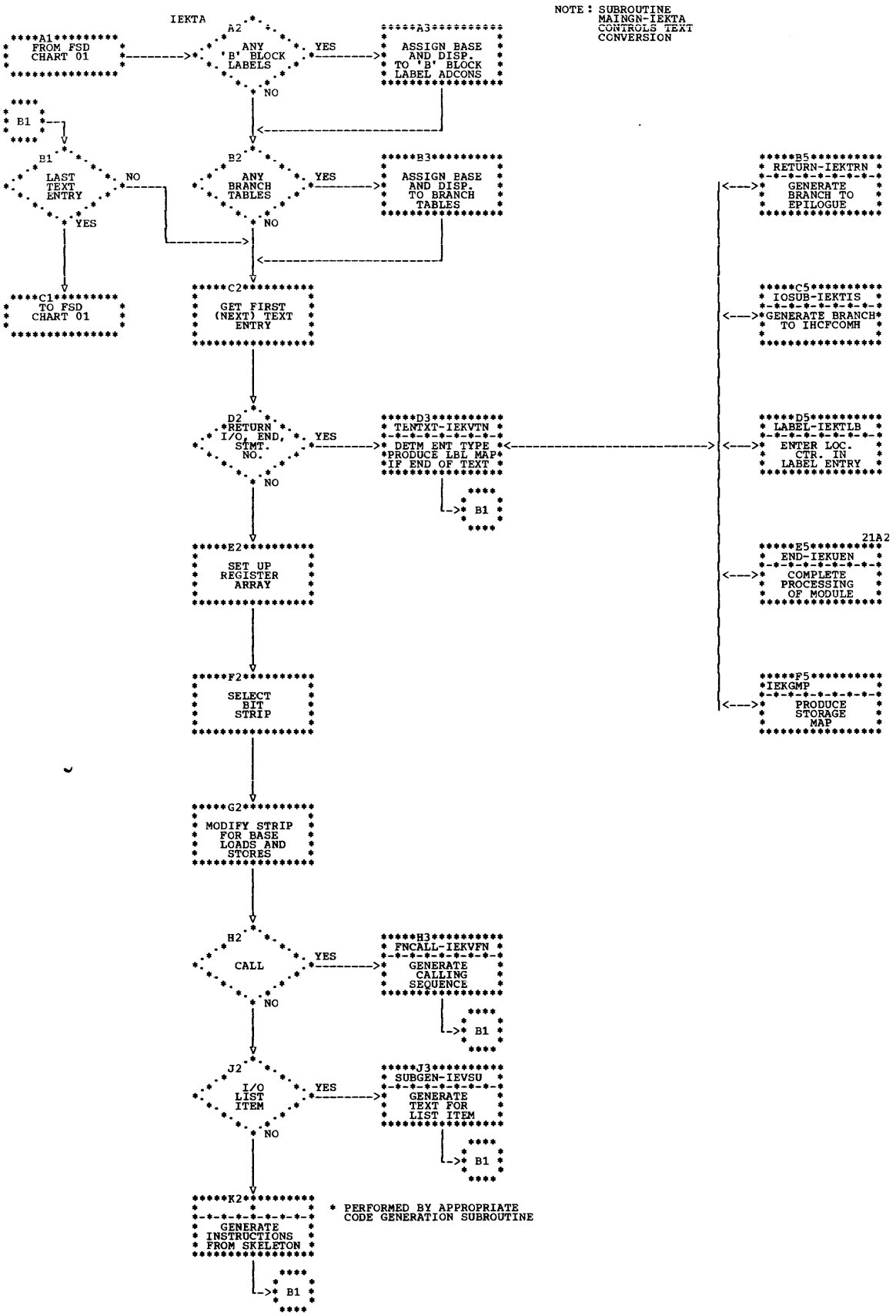


Chart 21. Subroutine END-IEKUEN

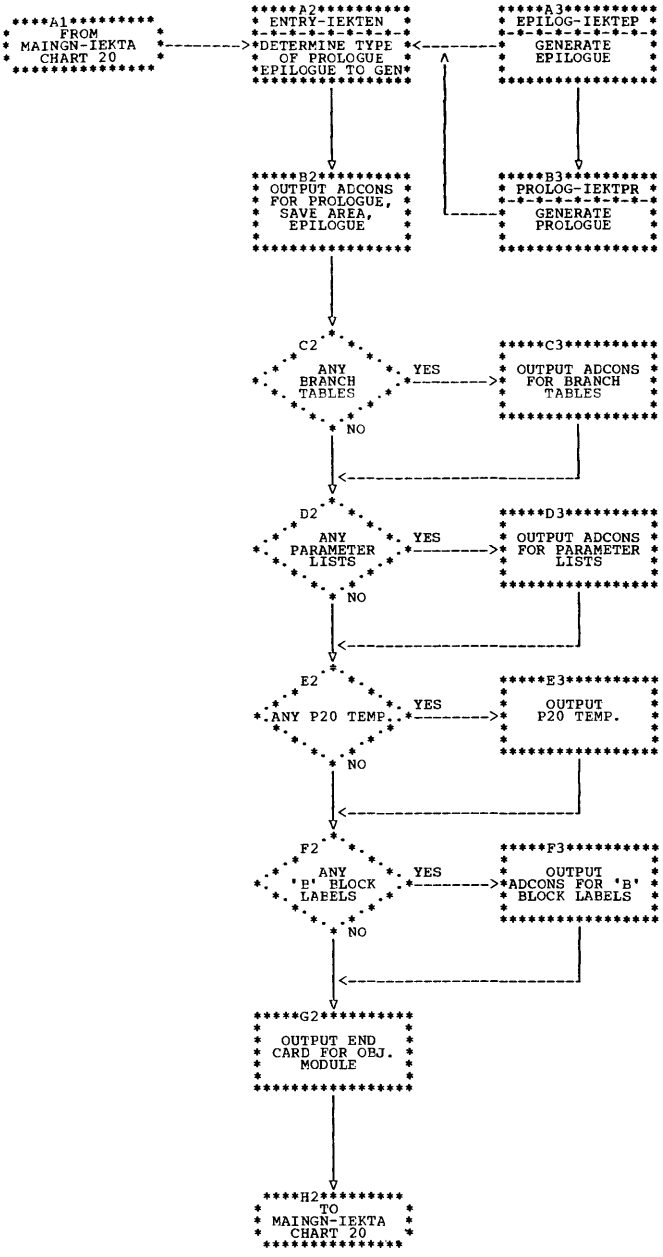


Table 14. Phase 25 Subroutine Directory (Part 1 of 2)

Subroutine	Function
ADMDGN-IEKVAD[1]	Generates instructions for the AMOD, DMOD, ABS, IABS, DABS, AND, OR, COMPL, LCOMPL, and DBLE in-line functions.
BITNFP-IEKVFP[1]	Generates instructions for the following text entries: BITON, BITOFF, BITFLP, TBIT, MOD24, SHFTR, and SHFTL in-line functions.
BRLGL-IEKVBL[1]	Generates instructions for the following text entries: Operator is a relational operator operating upon two operands or upon one operand and zero, assigned GO TO operators, computed GO TO operators, unconditional branching, branch true and branch false operations, and ASSIGN statement.
CGEN-IEKWCN	Common data area in which the arrays used during code generation are initialized.
END-IEKUEN	Performs final processing of the object module.
ENTRY-IEKTEN	Calls routines PROLOG-IEKTPR and EPILOG-IEKTEP to generate prologues and epilogues for subroutines and secondary entry points. Generates prologues and epilogues for the main program.
EPILOG-IEKTEP	Generates the epilogues associated with a subprogram and its secondary entry points (if any).
FAZ25-IEKP25	Common data area used by phase 25.
FNCALL-IEKVFN	Generates calling sequences for CALL statements (other than those to IHCFCOMH) and function references. Generates the instructions to store the result returned by a function subprogram.
GOTOKK-IEKWKK	Used by subroutine MAINGN-IEKTA to branch to the code generation subroutines.
IOSUB-IEKTIS/ IOSUB2-IEKTIO	Generates calling sequences for calls to IHCFCOMH.
LABEL-IEKTLB	Processes statement numbers by entering the current value of the location counter into the statement number entry in the dictionary.
LISTER-IEKTLS	Produces a listing of the final compiler-generated instructions.
MAINGN-IEKTA/ MAINGN2-IEKVM2	Assign base and displacement for 'B' block label adcons and branch tables. Control the text conversion process of phase 25.
PACKER-IEKTPK	Packs the various parts of each instruction produced during code generation into a TXT record.
PLSGEN-IEKVPL[1]	Generates the instructions for the following text entries: real multiplication and division operations, addition and subtraction operations, half- and full-word integer multiplication, half- and full-word integer division, and MOD in-line function.
PROLOG-IEKTPR	Generates prologues for subroutines and secondary entry points (if any).
RETURN-IEKTRN	Processes the RETURN statement by generating a branch to the epilogue.
STOPPR-IEKTSR	Generates character strings in calls to IHCFCOMH for STOP and PAUSE statements.

[1]Code generation subroutines.

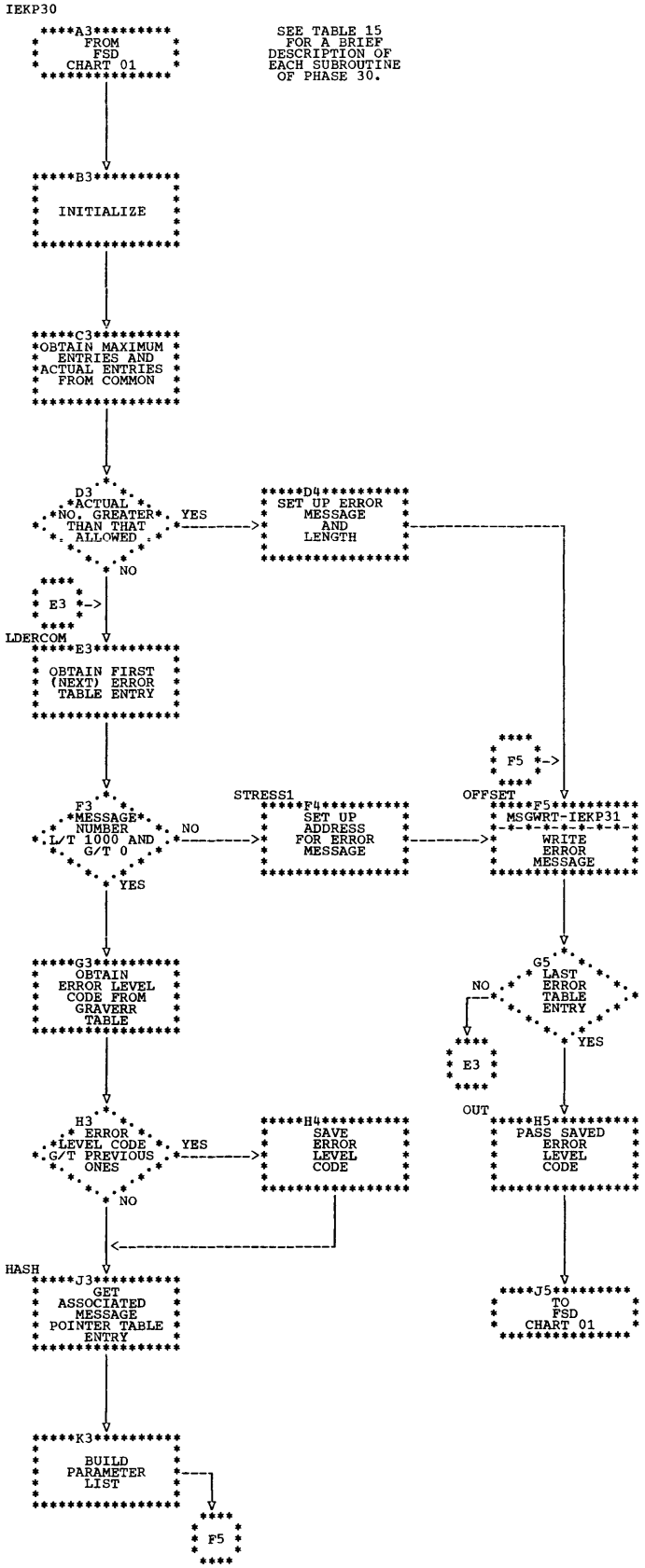
Table 14. Phase 25 Subroutine Directory (Part 2 of 2)

Subroutine	Function
SUBGEN-IEKVSU[1]	Generates instructions for the following text entries: subscript operations, right and left shift operations, store operations, and list item operations.
TENTXT-IEKVTN	Controls the processing of END, RETURN, and input/output statements, statement numbers, and end of I/O list indicators. Produces label map.
TSTSET-IEKVTS[1]	Generates the instructions to (1) compare two operands across a relational operator, and (2) set operand 1 to either true or false depending upon the outcome of the comparison. Generates the following in-line functions: FLOAT, DFLOAT, INT, IDINT, IFIX, HFIX, DIM, IDIM, SIGN, ISIGN, DSIGN, MAX2, and MIN2.
UNRGEN-IEKVUN[1]	Generates the instructions for the following text entries: unary minus operations (e.g., A=-B), logical NOT operations, load byte operations, load address operations, AND, OR, and XOR operations.
IEKGMP	Produces a storage map.
[1]Code generation subroutines.	

Table 15. Phase 30 Subroutine Directory

Subroutine	Function
IEKP30	Controls phase 30 processing.
MSGWRT-IEKP31	Writes the error messages using the FSD.

Chart 22. Phase 30 (IEKP30) Overall Logic



This appendix contains text and figures that describe and illustrate the major tables used and/or generated by the FORTRAN System Director and the compiler phases. The tables are discussed in the order in which they are generated or first used. In addition, table modifications resulting from the compilation process are explained, where appropriate, after the initial formats of the tables have been explained.

COMMUNICATION TABLE (NPTR)

The communication table (referred to as the NPTR table in the program listing), as a portion of the FORTRAN System Director, resides in main storage throughout the compilation. It is a central gathering area used to communicate necessary information among the various phases of the compiler.

Various fields in the communication table are examined by the phases of the compiler. The status of these fields determines:

- Options specified by the source programmer.
- Specific action to be taken by a phase.

If the field in question is null, the option has not been specified or the action is not to be taken. If the field is not null, the option has been specified or the action is to be taken. Table 16 illustrates the organization of the communication table.

CLASSIFICATION TABLES

Classifying, a function of the preparatory subroutine (GETCD-IEKCGC) of phase 10, involves the assignment of a code to each type of source statement. This code indicates to the DSPTCH-IEKCDP subroutine which subroutine (either keyword or arithmetic) is to continue the processing of that source statement. The following paragraph describes the processing that occurs during classifying. The tables used in the classifying process are the keyword pointer (IPTR) and the keyword table (ITBLE), which exist in GETCD-IEKCGC. They are illustrated in Tables 17 and 18, respectively.

The source statement might be classified during source statement packing if the statement classification is one of those listed in Table 19. For example, an arithmetic statement would be assigned the code 56 (see note). Otherwise, the classifying process determines the type of the source statement by comparing the first character of the packed source statement with each character in the keyword pointer table. If that first character corresponds to the initial character of any keyword, the keyword pointer table is then used to obtain a pointer to a location in the keyword table. This location is the first entry in the keyword table for the group of keywords beginning with the matched character. All characters of the source statement, up to the first delimiter, are then compared with that group of keywords.If a match results, the classification code associated with the matched entry is assigned to the source statement. If a match does not result, or if the first character of the source statement does not correspond to the first character of any of the keywords, the source statement is classified as an invalid statement.

Note: The packing process, which precedes classifying, marks a source statement as arithmetic if, in that statement, an equal sign that is not bounded by parentheses is encountered. If the source statement has been marked as arithmetic, it is classified accordingly by the classification process.

Table 16. Communication Table [NPTR(2,36)] (Part 1 of 2)

	1 (4 bytes)	2 (4 bytes)
1	Relative location of temporary for FLOAT/FIX (CORAL, phase 25)	Pointer to 1-character symbol chain
2	Previous classification code (phase 10); register currently assigned (phase 20, OPT=0 only)	Pointer to 2-character symbol chain
3	Options: DUMP, XL, XREF, ID, EDIT, MAP, LOAD, DECK, LIST, BCD, SOURCE	Pointer to 3-character symbol chain
4	Pointer to most recently generated EQUIVALENCE group entry (phase 10); relative location of first temporary (CORAL, phase 25).	Pointer to 4-character symbol chain
5	Current NADCON index (PHAZ15); NADCON index for first adcon (CORAL); NADCON index for first temporary (phase 20, 25).	Pointer to 5-character symbol chain
6	Maximum line count	Pointer to 6-character symbol chain
7	NADCON index for last statement number	Pointer to last dictionary entry in stmt number chain (XREF--phase 10); number of reserved registers set aside for data plus RX branching, in addition to register 13 (phase 20 prior to Branching Optimization, OPT ≠ 0, optimization not downgraded); number of reserved registers used for data plus RX branching, in addition to registers 13 & 12 (phase 25, OPT ≠ 0, optimization not downgraded).
8	Type of text (phase 10); pointer to next phase 10 text item (PHAZ 15); pointer to .TXX or .QXX temporary chain (phase 20); text creation indicator: set to 254 during processing of a case 2 subscript which requires an adcon text item to be inserted before (phase 20, OPT=0 only).	
9	Pointer to next available text entry	Pointer to end of text.
10	Name of routine (subprogram/main program)	
11	Phase in control indicator	Trace switch; optimization downgrade switch-bit 13 (PHAZ15, phase 20).
12	Index to last available error table entry.	
13	END card indicator (phase 10)	Pointer to first card of source pgm.
14	Relative location of parameter lists (PHAZ15, phase 25)	Pointer to 4-byte constant chain
15	NADCON index for 1st parameter list (PHAZ15, phase 25)	Pointer to 8-byte constant chain
16	Page count	Pointer to 16-byte constant chain
17	Current line count	Pointer to statement number chain

Table 16. Communication Table [NPTR(2,36)] (Part 2 of 2)

	1	2
18	Relative location for register 13	Number of branch table entries (STALL); relative location for register 12.
19	Active base register: 0 for reg. 13, 4096 for reg. 12.	NADCON index for first temporary (phase 20).
20	Secondary entry points if nonzero	Number of times XREF buffer has been written out (phase 10); pointer to temporary used for subscript index evaluation (Register Optimization).
21	Location counter, except in phase 20 (other than branching optimization) where it is relative location for active base register	NADCON index for first COMMON area
22	Pointer to dictionary entry for IBCOM	Index to next available error table entry
23	External function and/or CALL indicator	Pointer to end of stmt. number chain (STALL)
24	Program uses FLOAT/FIX or MOD function if nonzero; arithmetic interrupt indicator; text optimization (IBCOM); pointer to .SXX temporary chain (phase 20, OPT=0 only).	Optimization level
25	Pointer to first dictionary entry	Pointer to COMMON chain
26	Pointer to DEFINE FILE text (phase 10); relative location of DEFINE FILE parameter lists (CORAL, phase 25).	Pointer to EQUIVALENCE chain
27	Pointer to literal constant chain	Pointer to data text chain; subscript of required skeleton (phase 25).
28	Pointer to DIOCS entry	Pointer to normal text chain
29	Pointer to branch table chain	Pointer to next available information table entry
30	BLOCK DATA subprogram switch	Pointer to end of information table
31	FUNCTION SUBPROGRAM switch	SUBROUTINE SUBPROGRAM switch
32	Pointer to namelist text chain; local variable (phase 25).	Pointer to format text chain; local variable (phase 25)
33	Size of constants; relative location of epilogue (CORAL, phase 25).	Size of variables; relative location of epilogue (CORAL, phase 25).
34	Current displacement from active reg. (phase 20)	NADCON index for first adcon (CORAL); current NADCON index (phase 20).
35	Relative location of adcon for first statement number; branching optimization (phase 25).	Delete/error switch
36	Number of source statements	

Table 17. Keyword Pointer Table (IPTR)

Character (1 byte)	Number[1] (1 byte)	Displacement[2] (2 bytes)
A	2	0
B	2	12
C	5	34
D	8	84
E	5	175
F	3	220
G	1	244
H	0	0
I	3	250
J	0	0
K	0	0
L	2	286
M	1	312
N	2	318
O	0	0
P	3	336
Q	0	0
R	5	357
S	3	399
T	2	428
U	0	0
V	0	0
W	1	447
X	0	0
Y	0	0
Z	0	0

[1]This field contains the number of keywords beginning with the associated character.
[2]This field contains the displacement from the beginning of the keyword table for the group of keywords associated with the character.

Table 18. Keyword Table (ITBLE) (Part 1 of 2)

Length-1[1]	Key Word[2]	Code[3]
5	ASSIGN	1
1	AT	9
8	BACKSPACE	2
8	BLOCKDATA	3
7	CONTINUE	5
5	COMMON	7
3	CALL	8
14	COMPLEXFUNCTION	4
6	COMPLEX	6
8	DIMENSION	14
3	DATA	17
22	DOUBLEPRECISIONFUNCTION	10
14	DOUBLEPRECISION	11
1	DO	18
9	DEFINEFILE	13
6	DISPLAY	15
4	DEBUG	16
10	EQUIVALENCE	19
6	ENDFILE	21
3	END (followed by group mark)[4]	23
4	ENTRY	22
7	EXTERNAL	20
5	FORMAT	25

[1]This part of the entry for each keyword is one byte in length and contains a value equal to the number of characters in that keyword minus one.
[2]This part of the entry for each keyword contains an image of that keyword at one byte per character.
[3]This part of the entry for each keyword is one byte in length and contains the classification code for that keyword.
[4]Represented in hexadecimal as '4F'

Table 18. Keyword Table (Part 2 of 2)

Length-1[1]	Key Word[2]	Code[3]
7	FUNCTION	24
3	FIND	12
3	GOTO	27
7	IMPLICIT	29
14	INTEGERFUNCTION	28
6	INTEGER	30
14	LOGICALFUNCTION	33
6	LOGICAL	35
3	MOVE	34
7	NAMELIST	36
5	NORMAL	37
4	PAUSE	38
4	PRINT	39
4	PUNCH	40
3	READ	44
5	RETURN	43
5	REWIND	42
11	REALFUNCTION	41
3	REAL	45
3	STOP	48
9	SUBROUTINE	46
8	STRUCTURE	47
7	TRACEOFF	49
6	TRACEON	50
4	WRITE	51

[1]This part of the entry for each keyword is one byte in length and contains a value equal to the number of characters in that keyword minus one.
[2]This part of the entry for each keyword contains an image of that keyword at one byte per character.
[3]This part of the entry for each keyword is one byte in length and contains the classification code for that keyword.

Table 19. Classification Codes Assigned During Source Statement Packing

Statement Classification/Condition	Code[1]
Logical IF	31
Arithmetic IF	32
Arithmetic	56
Excessive continuation cards	57
Unclassifiable	59
Unbalanced parentheses	61
Bad label	62

[1]These codes are not in the keyword tables.

NADCON TABLE

The NADCON table, built by PHAZ15 and CORAL and partially overwritten by phase 20, contains:

1. Parameter list pointers.
2. Adcons for local variables and constants.
3. Adcons for variables in COMMON and for those equivalenced into COMMON.
4. Adcons for external references.

The information in the table is used by CORAL and phase 25. Each table entry is one word in length; the format of the table is shown in Table 20.

Table 20. NADCON Table

Parameter list pointer entries (one word per entry)
Adcon entries for local variables and constants (one word per entry)
Adcon entries for variables in COMMON and those equivalenced into COMMON (one word per entry)
Adcon entries for external references (one word per entry)

Parameter entries are created by PHAZ15. Each entry is a pointer to the dictionary entry for the parameter. Indicators denote ends of parameter lists and also parameters shared by more than one function or subroutine call.

Adcon entries are created by CORAL and then inserted by CORAL into the adcon portion of the object module (see Figure 9). Pointers to temporaries are created by phase 20 and placed in the portion of the table used previously by CORAL.

Phase 25 inserts the parameters and temporaries into the object module. The right-hand portion of Figure 9 indicates the sequence in which storage is assigned in the object module and the data which is entered into that storage.

INFORMATION TABLE

The information table (referred to as NDICT or NDICTX) is constructed by Phase 10 and modified by subsequent phases. This table contains entries that describe the operands of the source module. The information table consists of five components: dictionary, statement number/array table, common table, literal table, and branch table.

INFORMATION TABLE CHAINS

The information table is arranged as a number of chains. A chain is a group of related entries, each of which contains a pointer to another entry in the group. Each chain is associated with a component of the information table.

The information table can contain the following chains:

- A maximum of nine dictionary chains: one for each allowable FORTRAN variable length (1 through 6 characters) and one for each allowable FORTRAN constant size (4, 8, or 16 bytes). Each dictionary chain for variables contains entries that describe variables of the same length. Each dictionary chain for constants contains entries that describe constants of the same size.
- One statement number/array chain for entries that describe statement numbers.
- Two common table chains: one for entries describing common blocks and their associated variables, and one for entries describing equivalence groups and their associated variables.
- One literal table chain for entries that describe literal constants used as arguments in CALL statements.
- One branch table chain composed of entries for statement numbers appearing in computed GO TO statements.

Entries describing the various operands of the source module are developed by Phase 10 and placed into the information table in the order in which the operands are encountered during the processing of the source module. For this reason, a particular chain's entries may be scattered throughout the information table and entries describing different types of operands may occupy contiguous locations within the information table. Figure 10 illustrates this concept.

CHAIN CONSTRUCTION

The construction of a chain requires: (1) initialization of the chain, and (2) pointer manipulation. Chain initialization is a two-step process:

1. The first entry of a particular type (e.g., an entry describing a variable of length one) is placed into the information table at the next available location.
2. A pointer to this first entry is placed into the communication table entry (see "Communication Table") reserved for the chain of which this first entry is a member.

Subsequent entries are linked into the chain via pointer manipulation, as described in the following paragraphs.

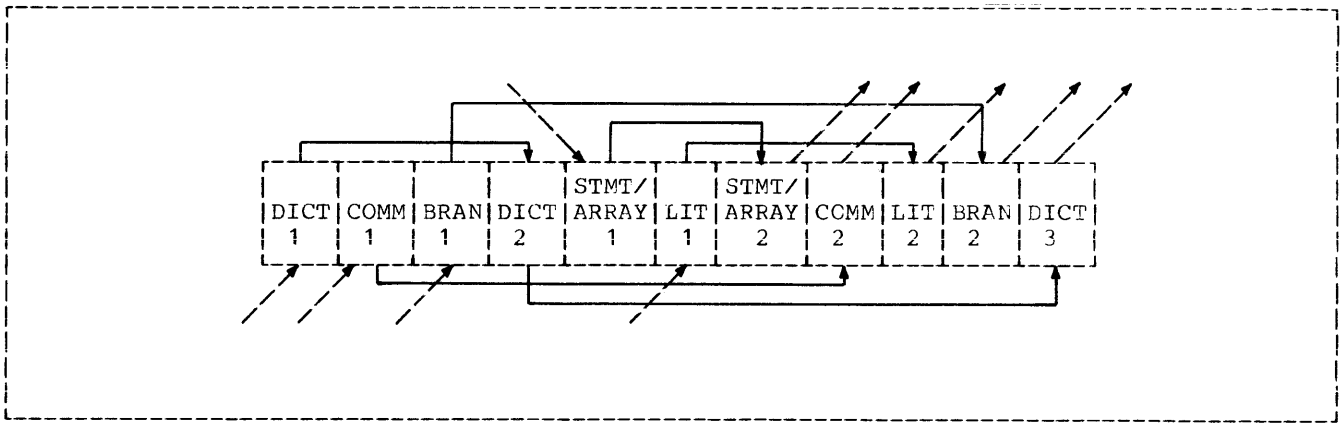


Figure 10. An Example of Information Table Chains

The communication table entry containing the pointer to the initial entry in the chain is examined and the first entry in the chain is obtained. The item that is to be entered is compared to the initial entry. If the two are equal, the item is not re-entered; if they are unequal, the first entry in the chain is checked to see if it is also the last. (An entry is the last in a chain if its "chain" field is zero.)

If the chain entry under consideration is the last in the chain, the new item is entered into the information table at the next available location, and a pointer to its location is placed into the chain field of the last chain entry. The new entry is thereby linked into the chain and becomes its last member.

If the entry under consideration is not the last in the chain, the next entry is obtained by using its chain field. The item to be entered is compared to the entry that was obtained. If the two are equal, the item is not re-entered; if they are unequal, the entry under consideration is checked to see if it is the last in the chain; etc.

This process is continued until a comparable entry is found or the end of the chain is found. If a comparable entry is found, the item is not reentered. If the new item is not found in the chain, it is then linked into the chain.

OPERATION OF INFORMATION TABLE CHAINS

The following paragraphs describe the operation of the various chains in the information table.

Dictionary Chain Operation

The operation of a dictionary chain is based upon "balanced tree" notation. This notation provides two chains, high and low (with a common midpoint), for the entries describing variables of the same length or constants of the same size. The initial midpoint is the first entry placed into the information table for a variable of a particular length or a constant of a particular size. When two entries have been made on the high side of the midpoint, the first entry on the current midpoint's high-chain becomes the new midpoint. Similarly, when two entries have been made on the low side of the midpoint, the first entry on the current midpoint's low-chain becomes the new midpoint.

A change of midpoint for a variable of a particular length or a constant of a particular size causes a pointer to the new midpoint to be recorded in the communication table. The following example illustrates the manner in which phase 10 employs the balanced tree notation to construct a dictionary chain.

Assume that the following variables appear in the source module in the order presented.

D C E F A B

When phase 10 encounters the variable D, it constructs a dictionary entry for it (see "Dictionary"), places this entry at the next available location in the information table, and records a pointer to that entry into the appropriate field of the communication table (see "Communication Table"). The entry for D is the initial midpoint for the chain of entries describing variables of length one. (When a dictionary entry is placed into the

information table, both the high- and low-chain fields of that entry are zero.)

When phase 10 encounters the variable C, it constructs a dictionary entry for it. Phase 10 then obtains the dictionary entry that is the initial midpoint and compares C to the variable in that entry. If the two are unequal, phase 10 determines whether or not the variable to be entered is greater than or less than the variable in the obtained entry. In this case, C is less than D in the collating sequence, and, therefore, phase 10 examines the low-chain field of the obtained entry, which is that for D. This field is zero, and the end of the chain has been reached. Phase 10 places the entry for C into the next available location in the information table and records a pointer to that entry in the low-chain field of the dictionary entry for D. The entry for C is thereby linked into the chain.

When the variable E is encountered, phase 10 carries out essentially the same procedure; however, because E is greater than D, phase 10 examines the high-chain field of the entry for D. It is zero, which denotes the end of the chain. Therefore, phase 10 places the dictionary entry for E into the next available location in the information table and records a pointer to that entry in the high-chain field of the dictionary entry for D.

When the variable F is encountered, phase 10 constructs a dictionary entry for it and compares it to the variable in the entry that is the common starting point for the chain. Because F is greater than D, phase 10 examines the high-chain field of the entry for D. This field is not zero and, hence, the end of the chain has not yet been reached. Phase 10 obtains the entry (for E) at the location pointed to by the nonzero chain field (of the entry for D) and compares F to the variable in the obtained entry. The variable F is greater than the variable E. Therefore, phase 10 examines the high-chain field of the entry for E. This field is zero and the end of the chain has been reached. Phase 10 places the entry for F into the next available location in the information table and records a pointer to that entry in the high-chain field of the entry for E. Since two entries have now been made on the high side of the current midpoint, the first variable on D's high-chain becomes the new midpoint.

Phase 10 carries out similar procedures to link the entries for the variables A and B into the chain.

(If one of the comparisons made between a variable to be entered into the dictionary and a variable in an entry already in the dictionary results in a match, the variable has previously been entered and is not reentered.)

Figure 11 illustrates the manner in which the entries for the variables are chained after the entry for B has been linked into the chain.

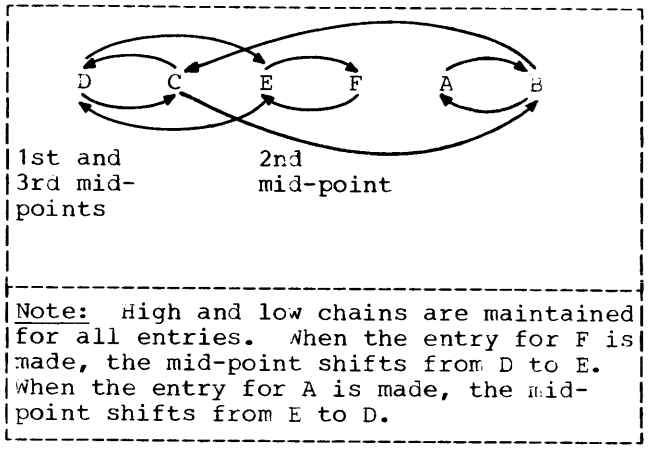


Figure 11. Dictionary Chain

Statement Number Chain Operation

The statement number chain constructed by phase 10 is linear; that is, each statement number entry (see "Statement Number/Array Table") is pointed to by the chain field of the previously constructed statement number entry. The first statement number entry is pointed to by a pointer in the communication table.

To construct the statement number chain, phase 10 places the statement number entry constructed for the first statement number in the module into the next available location in the information table. It records a pointer to that entry in the appropriate field of the communication table. (When a statement number entry is placed into the information table, its chain field is zero.) Phase 10 links all other statement number entries into the chain by scanning the previously

constructed statement number entries (in the sequence in which they are chained) until the last entry is found. The last entry is denoted by a zero chain field. Phase 10 then places the new entry at the next available location in the information table and records a pointer to that entry in the zero chain field of the last entry in the chain. The new entry is thereby linked into the chain and becomes its last member. (Throughout the construction of the statement number chain, phase 10 makes comparisons to insure that a statement number is entered only once.)

Common Chain Operation

The chain constructed by phase 10 due to COMMON statements appearing in the source module is bi-linear; that is, phase 10 links together:

1. The individual COMMON block name entries (see "COMMON Table") that it develops for the COMMON block names appearing in the module.

2. The dictionary entries (see "Dictionary") that it develops for the variables appearing in a particular common block. (The dictionary entry for the first variable appearing in a COMMON block is also pointed to by the COMMON block name entry for the COMMON block containing the variable.)

To construct the COMMON chain, phase 10 places the COMMON block name entry that it constructs for the first COMMON block name appearing in the module at the next available location in the information table. It records a pointer to this entry in the appropriate field of the communication table. Phase 10 then obtains the first variable in the COMMON block, constructs a dictionary entry for it, places the entry at the next available location in the information table, and records a pointer to that entry in the P1 and P2 field of the COMMON block name entry for the COMMON block containing the variable. Phase 10 obtains the next variable in the common block, constructs a dictionary entry for it, places the entry in the information table, records a pointer to that entry in the COMMON chain field of the dictionary entry constructed for the variable encountered immediately prior to the variable under consideration (this entry location is obtained from the P2 field of the COMMON block name entry), and records a pointer to the information table for the new COMMON variable in the P2 field. Thus, the P2 field of the COMMON block name entry always contains a pointer to the information table entry for the last variable of a given COMMON block. Phase 10 obtains the next variable in the COMMON block, etc.

When phase 10 encounters a second unique COMMON block name, it constructs a COMMON block name entry for it, places the entry in the information table, and records a pointer to that entry in the chain field of the last COMMON block name entry, which is found by scanning the chain of such entries until a zero chain field is detected. Phase 10 then links the dictionary entries that it constructs for the variables appearing in the second COMMON block into the chain in the previously described manner.

If a COMMON block name is repeated in the source module a number of times, phase 10 constructs a COMMON block name entry only for the first appearance. However, it does include as members of the COMMON block the variables associated with the second and subsequent mentions of the COMMON block name. Phase 10 constructs a dictionary entry for the first variable associated with the second mention of the COMMON block name and places it into the information table. It then records a pointer to the dictionary entry for the new variable in the COMMON chain field of the last variable associated with the first mention of the COMMON block name. Phase 10 links the dictionary entry it constructs for the second variable associated with the second mention of a COMMON block name to the dictionary entry for the first variable associated with the second mention of that name; etc.

If a third mention of a particular COMMON block name is encountered, phase 10 processes the associated variables in a similar manner. It links the dictionary entries constructed for these variables as extensions to the dictionary entries developed for the variables associated with the second mention of the COMMON block name.

Equivalence Chain Operation

The chain constructed by phase 10 due to EQUIVALENCE statements appearing in the source module is also bi-linear. Phase 10 links together:

1. The individual equivalence group entries (see "COMMON Table") that it constructs for the equivalence groups appearing in the module.

2. The equivalence variable entries (see "COMMON Table") that it constructs for the variables appearing in a particular equivalence group. (The equivalence variable entry for the first variable appearing in an equivalence group is pointed to by the equivalence group entry for the group containing the variable.)

The construction of the equivalence chain by phase 10 parallels its construction of the COMMON chain. It links the equivalence group entries in the same manner as it does COMMON block name entries, and links equivalence variable entries in the same manner as the dictionary entries for the variables in a COMMON block. (The location of the last EQUIVALENCE group entry generated is recorded in the appropriate field of the communication table; the location of the last EQUIVALENCE variable entry generated is recorded locally in the keyword subroutine that processes the EQUIVALENCE statement).

Literal Constant Chain Operation

The chain constructed by phase 10 for the literal constant information appearing in the source module is linear. The literal constants are chained in reverse order of occurrence. Phase 10 records a pointer to the most recent literal constant entry generated. As each new entry is made, it is chained to the previous entry and it, in turn, is recorded as the most recent.

Branch Table Chain Operation

The phase 10 construction of the branch table chain parallels that of the statement number chain. It records a pointer to the first branch table entry (see "Branch Table") that is placed into the information table in the appropriate field of the communication table. In the chain field of the previously developed branch table entry, phase 10 records a pointer to the location in the information table for any new branch table entry. Unlike statement number entry processing, no label comparison is necessary. Thus, scanning the chain is avoided by recording the location of the last branch table entry in the P2 field of the first Initial Branch Table entry.

INFORMATION TABLE COMPONENTS

The following text describes the contents of each component of the information table and presents illustrations of phase 10 formats of the entries for each component. Modifications made to these entries by subsequent phases of the compiler are also illustrated.

Dictionary

The dictionary contains entries that describe the variables and constants of the source module. The information gathered for each variable or constant is derived from an analysis of the context in which the variable or constant is used in the source module.

VARIABLE ENTRY FORMAT: The format of the dictionary entries constructed by phase 10 for the variables of the source module is illustrated in Figure 12.

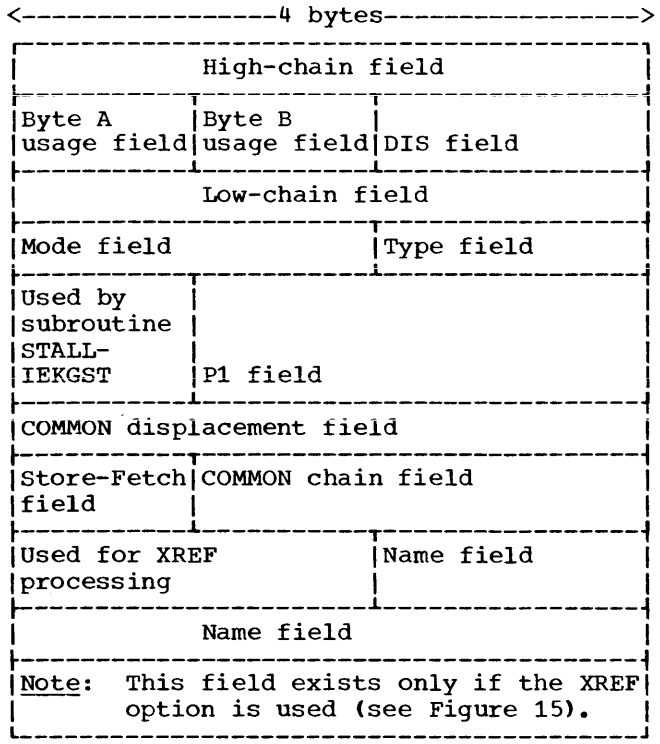


Figure 12. Format of Dictionary Entry for Variable

Subfield	Function
Bit 0 'on'	variable appears in a STRUCTURE statement
Bit 1 'on'	symbol referred to
Bit 2 'on'	variable is in COMMON
Bit 3 'on'	not used
Bit 4 'on'	variable is equivalenced
Bit 5 'on'	variable has appeared in an EQUIVALENCE group that has been processed by subroutine STALL-IEKGST (used by phase 15)
Bit 6 'on'	variable is an external subprogram name
Bit 7 'on'	variable appears in Type statement

Figure 13. Function of Each Subfield in the Byte A Usage Field of a Dictionary Entry for a Variable or Constant

High-Chain Field: The high-chain field is used to maintain linkage between the various entries in the chain. It contains either a pointer to an entry that collates higher in the collating sequence or an indicator (zero), which indicates that entries in the chain that collate higher than itself have not yet been encountered.

Byte A Usage Field: This field is contained in the first byte of the second word. This field indicates a portion of the characteristics of the variable for which the dictionaty entry was created. The byte A usage is divided into eight subfields, each of which is one bit long. The bits are numbered from 0 through 7. Figure 13 indicates the function of each subfield in the byte A usage field.

Byte B Usage Field: The byte B usage field is contained in the second byte of the second word. This field indicates additional characteristics of the variable entered into the dictionary. It is divided into eight subfields, each of which is one bit long. The bits are numbered from 0 through 7. Figure 14 illustrates the function of each subfield in the byte B usage field.

Subfield	Function
Bit 0 'on'	variable is "call by value" parameter
Bit 1 'on'	variable is "call by name" parameter
Bit 2 'on'	variable is used as an argument
Bit 3 'on'	variable has appeared in a previous DATA statement (phase 15)
Bit 4 'on'	not used
Bit 5 'on'	variable is used as a subscript
Bit 6 'on'	variable is in COMMON, or in an EQUIVALENCE group and has been assigned a relative address (phase 15)
Bit 7 'on'	variable appears in DATA statement

Figure 14. Function of Each Subfield in the Byte B Usage Field of a Dictionary Entry for a Variable

DIS Field: The DIS field contains either the displacement of a structured variable from the head of its structure group or the number of dummy arguments for a statement function name. If the variable is neither structured nor a statement function name, this field contains a count of the number of times the variable appears in the source program.

Low-Chain Field: The low-chain field is used to maintain linkage between the various entries in the chain. It contains either a pointer to an entry that collates lower in the collating sequence or an indicator (zero), which indicates that entries in the chain that collate lower than itself have not yet been encountered.

Mode/Type Field: The mode/type field is divided into two subfields, each two bytes long. The first two bytes (mode subfield) are used to indicate the mode of the variable (e.g., integer, real); the second two bytes (type subfield) are used to indicate the type of the variable (e.g., array, external function). Both the mode and type are numeric quantities and correspond to the values stated in the mode and type tables (see Tables 21 and 22).

P1 Field: The P1 field contains either a pointer to the dimension information in the statement number/array table if the entry is for an array (i.e., a dimensioned variable), or a pointer to the text generated for the statement function (SF) if the entry is for an SF name. If the entry is neither for the name of an array nor the name of a statement function, the field is zero.

COMMON Displacement Field: The displacement of the variable, if it is in COMMON, is placed in this field by phase 10. This information will be moved to the DIS field by CORAL and replaced with a pointer to the dictionary entry for its COMMON block.

Store-Fetch Field: The Store-Fetch field contains information about the variable. If the variable is stored into, bit 0=1; if the variable is fetched, bit 1=1.

Table 21. Operand Modes

Mode of Operand	Internal Representation (in hexadecimal)
No mode (e.g., base variables)	0
Logical*1	2
Logical*4	3
Integer*2	4
Integer	5
Real*8	6
Real*4	7
Complex*16	8
Complex*8	9
Literal	A
Statement number	B
Hexadecimal	C
Namelist	D
Repeat constant	F

Table 22. Operand Types

Type of Operand	Internal Representation (in hexadecimal)
Scalar	0
Dummy scalar	1
Array	2
Dummy array	3
External subprogram	4
Constant	5
Statement function	6
Negative scalar	8
Negative dummy scalar	9
Negative array	A
Negative dummy array	B
Negative external subprogram	C
Negative constant	D
Negative statement function	E
QXX temporary (created by text optimization)	F

COMMON Chain Field: This field is used to maintain linkages between the variables in a COMMON block. It contains a pointer to the dictionary entry for the next variable in the COMMON block. (If the variable for which a dictionary entry is constructed is not in COMMON, this field is not used.)

Name Field: This field contains the name of the variable (right-justified) for which the dictionary entry was created.

MODIFICATIONS TO DICTIONARY ENTRIES FOR VARIABLES: During compilation, certain fields of the dictionary entries for variables may be modified. The following examples illustrate the formats of dictionary entries for variables at various stages of phase 10 and phase 15 processing. Only changes are indicated; * stands for unchanged.

Dictionary Entry for Variable After Preparation for XREF Processing: The format of a dictionary entry for a variable after subroutine CSORN-IEKCCR processing is illustrated in Figure 15.

XREF Buffer Pointer -- Last Entry: This field contains a pointer to the most recent XREF buffer entry for the symbol.

XREF Buffer Count: This field contains a count of the number of times the XREF buffer has been written out on SYSUT2 at the time that this dictionary entry is modified by subroutine CSORN-IEKCCR.

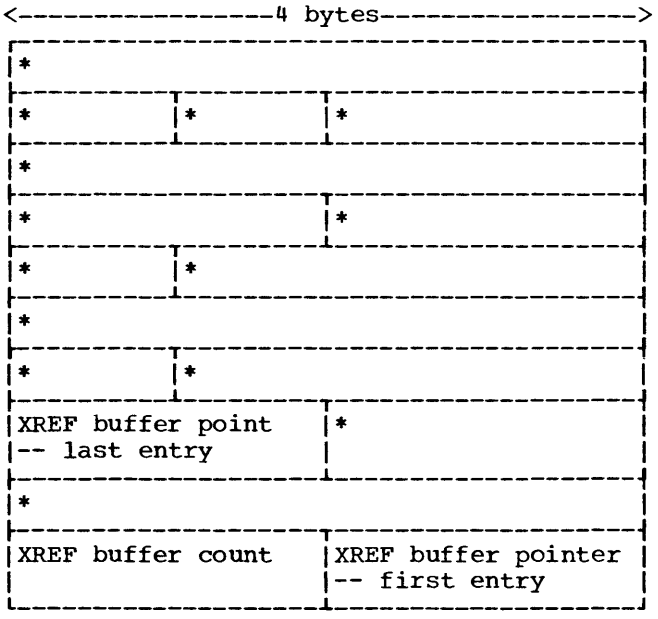


Figure 15. Format of Dictionary Entry for Variable After CSORN-IEKCCR Processing for XREF

XREF Buffer Pointer -- First Entry: This field contains a pointer to the first XREF buffer entry for this symbol.

Dictionary Entry for Variable After Co-ordinate Assignment: The format of a dictionary entry for a variable after co-ordinate assignment by the STALL-IEKGST subroutine is illustrated in Figure 16.

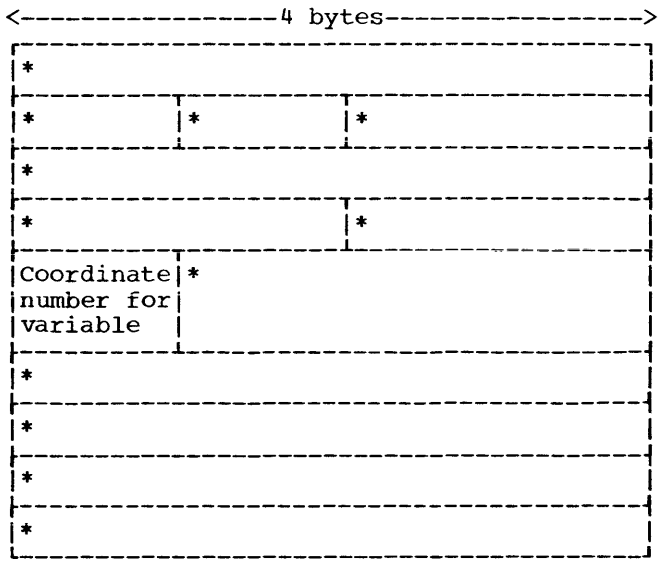


Figure 16. Format of Dictionary Entry for Variable After Coordinate Assignment

Dictionary Entry for Variable After COMMON Block Processing: The format of a dictionary entry for a variable after COMMON block processing is illustrated in Figure 17.

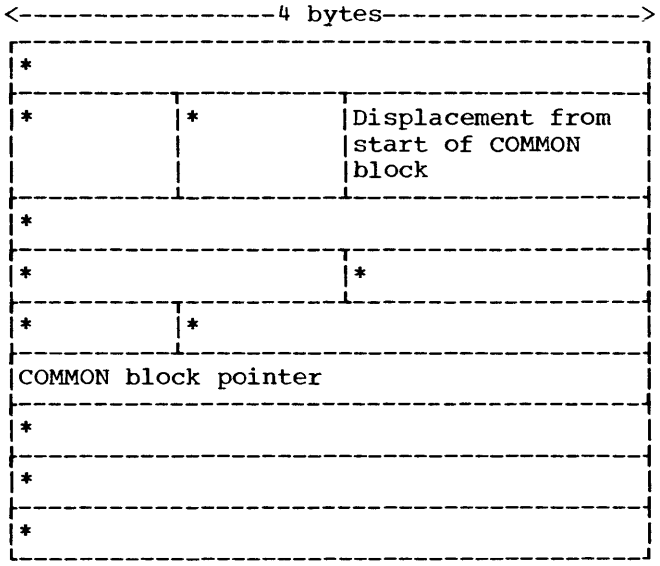


Figure 17. Format of Dictionary Entry for Variable After COMMON Block Processing

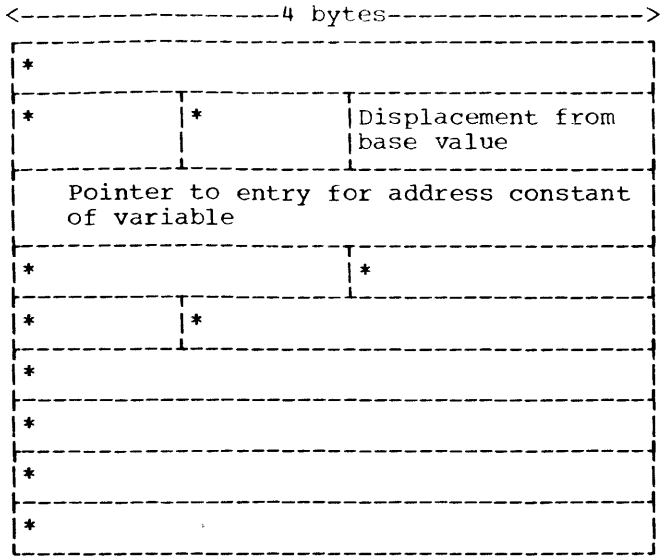


Figure 18. Format of Dictionary Entry for a Variable After Relative Address Assignment

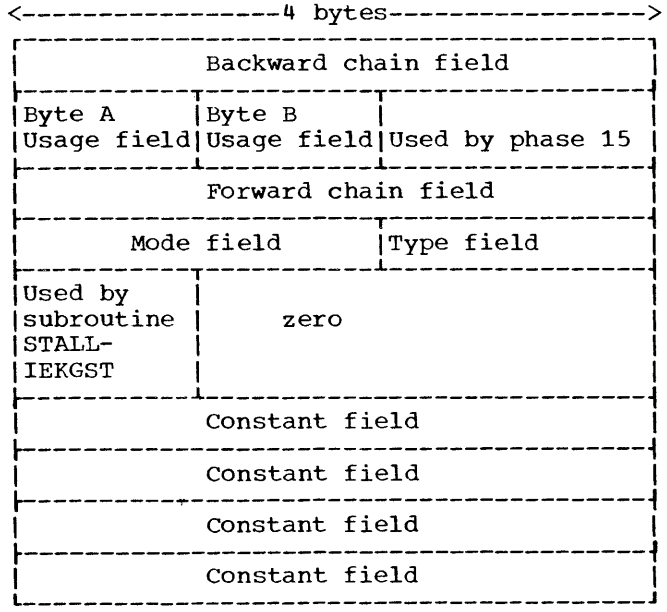


Figure 19. Format of Dictionary Entry for Constant

Dictionary Entry for Variable After Relative Address Assignment: The format of a dictionary entry for a variable after relative address assignment is illustrated in Figure 18.

CONSTANT ENTRY FORMAT: The format of the dictionary entries constructed by phase 10 for the constants of the source module is illustrated in Figure 19. It is similiar to that for a variable. The changes the entry undergoes during processing are the same except that a constant does not undergo XREF or COMMON processing. Also, for constants referred to implicitly, PHAZ15 sets a referenced bit to on. (Bit 1 in the byte A usage field; see Figure 13.)

Statement Number/Array Table

The statement number/array table contains statement number entries, which describe the statement numbers of the source module, and dimension entries, which describe the arrays of the source module.

STATEMENT NUMBER ENTRY FORMAT: The format of the statement number entries constructed by phase 10 is illustrated in Figure 20.

Chain Field: The chain field is used to maintain linkage between the various entries in the chain. It contains either a pointer to the next statement number entry in the chain or an indicator (zero), which indicates the end of the statement number chain.

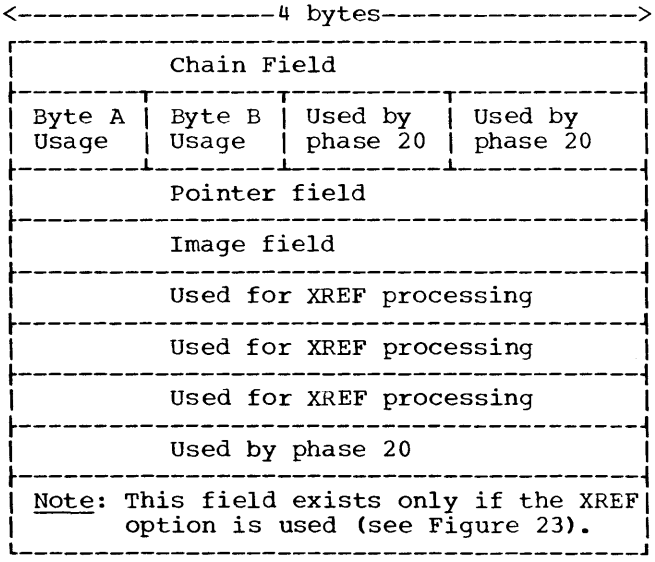


Figure 20. Format of a Statement Number Entry

Byte A Usage Field: This field is contained in the first byte of the second word. This field indicates a portion of the characteristics of the statement number for which the entry was created. The byte

A usage field is divided into eight subfields, each of which is one bit long. The bits are numbered from 0 through 7. Figure 21 indicates the function of each subfield of this field.

Byte B Usage Field: This field is contained in the second byte of the second word. The byte B usage field indicates additional characteristics of the statement number for which the entry was constructed. The byte B usage field is divided into eight subfields, each of which is one bit long. The bits are numbered 0 through 7. Figure 22 indicates the function of each subfield in the byte B usage field.

Pointer Field: If the entry is for the first statement number, this field contains a pointer to the last statement number entry. Otherwise, the field contains zeros.

Image Field: This field contains the binary representation of the statement number for which the entry was created.

Subfield	Function
Bit 0 'on'	statement number defined
Bit 1 'on'	statement number referred to
Bit 2 'on'	referred to in an ASSIGN statement
Bit 3	not used
Bit 4 'on'	statement number of a FORMAT statement
Bit 5 'on'	statement number of a GO TO, PAUSE, RETURN, STOP, or DO statement
Bit 6 'on'	statement number used as an argument
Bit 7 'on'	statement number is the object of a branch

Figure 21. Function of Each Subfield in the Byte A Usage Field of a Statement Number Entry

MODIFICATIONS TO STATEMENT NUMBER ENTRIES: During the processing of subroutines LABTLU-IEKCLT and STALL-IEKGST in phase 10, phases 15, 20, and 25, each statement number entry created by phase 10 is updated with information that describes the text block associated with the statement number. During phase 10, if the XREF option is selected, subroutine LABTLU-IEKCLT makes changes in statement number dictionary entries for later use by subroutine XREF-IEKXRF (see Figure 23).

Subfield	Function
Bit 0 'on'	statement number is within a DO loop and is transferred to from outside the range of the DO loop
Bit 1 'on'	compiler generated statement number
Bits 2-5	not used
Bit 6 'on'	statement number appears in END or ERR parameter of READ statement (branching optimization)
Bit 7 'on'	statement number is used in a computed GO TO statement

Figure 22. Function of Each Subfield in the Byte B Usage Field of a Statement Number Entry

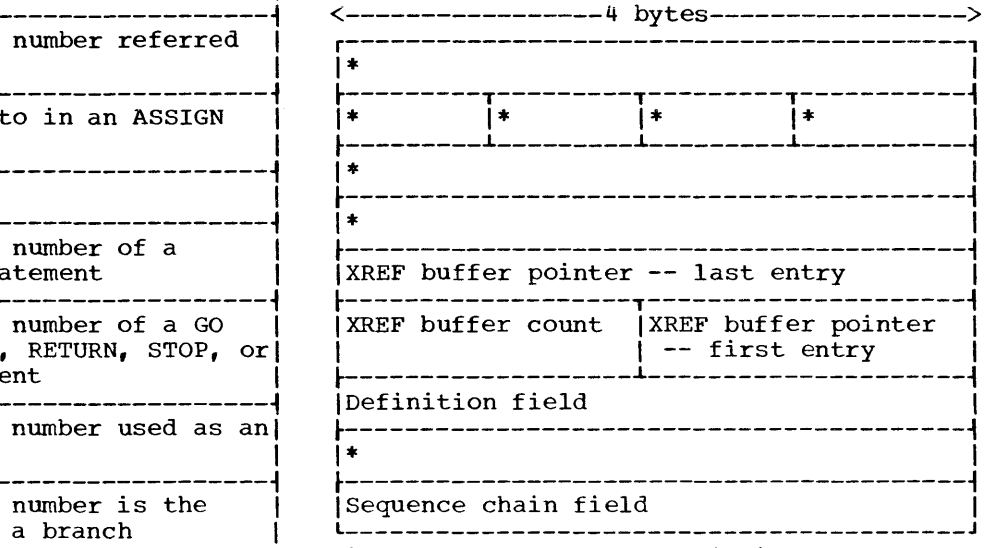


Figure 23. Format of a Dictionary Entry for Statement Number After Subroutine LABTLU-IEKCLT Processing for XREF

XREF Buffer Pointer -- Last Entry: This field contains a pointer to the most recent XREF buffer entry for this statement number, unless this dictionary entry is a definition of a statement number. If this dictionary entry is a definition of a statement number, this field is not used.

XREF Buffer Count: This field contains a count of the number of times the XREF buffer has been written out on SYSUT2 at the time this dictionary entry is modified by subroutine LABTLU-IEKCLT.

XREF Buffer Pointer -- First Entry: This field contains a pointer to the first XREF buffer entry for this statement number.

Definition Field: This field contains an ISN if this statement number dictionary entry corresponds to a definition of a statement number. The field contains -1 if the statement number has been previously defined.

Sequence Chain Field: This field chains the statement numbers in numerical order.

Figure 24 illustrates the format of a statement number entry after the processing of the STALL-IEKGST subroutine and phases 15, 20, and 25. Only changes are indicated; * stands for unchanged.

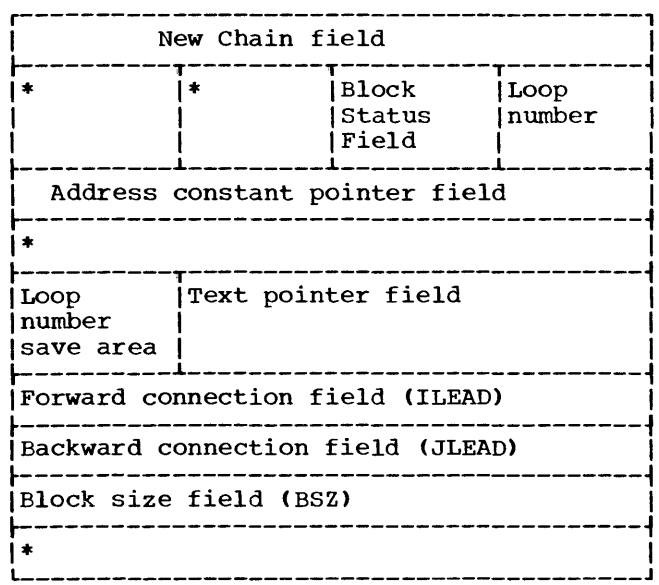


Figure 24. Format of Statement Number Entry After the Processing of Phases 15, 20, and 25

New Chain Field: The new chain field contains a pointer to the entry for a statement number. The number is the one that is defined in the source module immediately after the statement number for which the statement number entry under consideration was constructed. (The STALL-IEKGST subroutine modifies the phase 10 chain pointer when it rechains the statement number entries to correspond to the order in which statement numbers are defined in the source module.) This field is not modified by subsequent phases.

Block Status Field: The block status field indicates the status of the text block associated with the statement number entry under consideration. The block status field is divided into eight subfields, each of which is one bit long. The bits are numbered 0 through 7. Figure 25 indicates the function of each subfield in the block status field.

Subfield	Function
Bit 0 Bit 1	Used for various reasons by the routines that explore connections (e.g., the associated block has previously been considered in the search for the back dominator of the block)
Bit 2 'on'	the associated block exits from a loop
Bit 3 'on'	the associated block is a fork (i.e., it has two or more forward connections)
Bit 4	same as bits 0 and 1
Bit 5 'on'	the associated block is in the current loop
Bit 6 'on'	the associated block has been completely processed along the OPT=2 path
Bit 7 'on'	the associated block is an entry block

Figure 25. Function of Each Subfield in the Block Status Field

Loop Number Field: The loop number field contains the number of the loop to which the text block (associated with the statement number entry under consideration) belongs. This field is set up and used by phase 20. Just before the loop number is assigned, this field contains a depth number.

Back Dominator Field: The back dominator field contains a pointer to the statement number entry associated with the back dominator of the text block associated with the statement number entry under

consideration. This field, set up and used by phase 20, occupies the address constant pointer field.

Address Constant Pointer Field: The address constant pointer field (after phase 25 processing) contains either of the following:

- An indication of a reserved register and a displacement of the address constant for the statement number (see Phase 25, "Address Constant Reservation").
- Zero, if:
  1. unreferenced
  2. referenced, but not by END or ERR parameter of a READ statement, and within range of a reserved register.

Text Pointer Field: The text pointer field contains a pointer to the phase 15 text entry for the statement number with which the statement number entry under consideration is associated. This field is not used by phase 10; it is filled in by phase 15, and is unchanged by subsequent phases.

Forward Connection Field (ILEAD): The forward connection field contains a pointer to the initial RMAJOR entry for the blocks to which the text block associated with the statement number entry under consideration connects. This field is set up by phase 15 and used by phase 20. The base and displacement of the block are stored in this field by phase 20, Branching Optimization.

Backward Connection Field (JLEAD): The backward connection field contains a pointer to the initial CMAJOR entry for the blocks that connect to the text block associated with the statement number entry under consideration. This field is set up by phase 15 and used by phase 20. During phase 25 the relative location of the block is stored in the field.

Block Size Field (BSZ): The block size field contains the number of bytes of code generated in the block associated with the statement number entry under consideration. It does not include the padding for the first occurence in the block of required boundary alignment. This field is set up and used by phase 20, Branching Optimization. The following flags are set in this field:

bit 1, for branch true or false in this block;
bit 4, if block is B-block;
bit 5, if block ends with branch other than computed GO TO;
bit 8, if the conditional NOP follows an even number of half-words in the block;
bit 9, for conditional NOP in this block.

DIMENSION ENTRY FORMAT: The format of the dimension entries constructed by phase 10 is illustrated in Figure 26.

Array Size Field: The array size field contains either the total size of the associated array or zero, if the array has variable dimensions.

Dimension Number Field: The dimension number field contains the number of dimensions (1 through 7) of the associated array.

Element Length Field: The element length field contains the length of each element (first dimension factor) in the associated array.

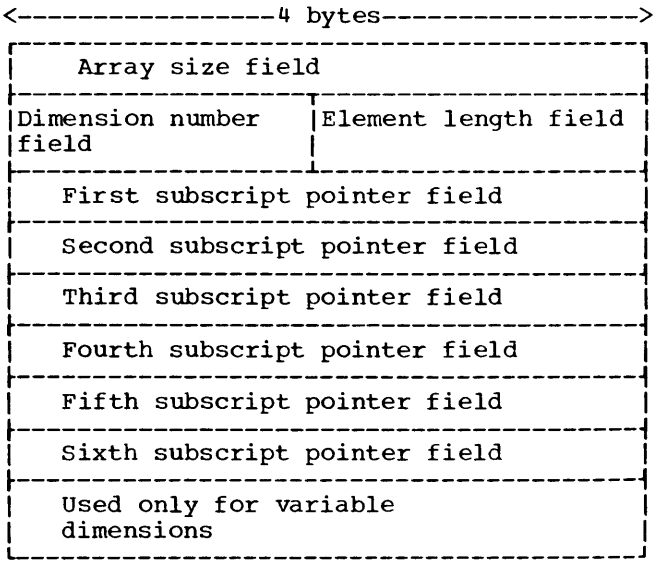


Figure 26. Format of Dimension Entry

First Subscript Pointer Field: The field contains either a pointer to the dictionary entry for the second dimension factor, which has a value of D1*L (see "Appendix F: Address Computation for Array Elements"), or a pointer to the dictionary entry for the first subscript parameter used to dimension the associated array if that array has variable dimensions. This field is not used if the associated array has a single non-variable dimension.

Second Subscript Pointer Field: This field contains either a pointer to the dictionary entry for the third dimension factor, which has a value of D1*D2*L, or a pointer to the second subscript parameter used to dimension the associated array if that array has variable dimensions. This field is not used if the associated array has a single dimension, or has two non-variable dimensions.

Third Subscript Pointer Field: This field contains either a pointer to the dictionary entry for the fourth dimension factor, which has a value of D1*D2*D3*L, or a pointer to the third subscript parameter used to dimension the associated array if that array has variable dimensions. This field is not used if the associated array has fewer than three dimensions, or has three non-variable dimensions.

Fourth Subscript Pointer Field: This field contains either a pointer to the dictionary entry for the fifth dimension factor, which has a value of D1*D2*D3*D4*L, or a pointer to the dictionary entry for the fourth subscript parameter used to dimension the associated array if that array has variable dimensions. This field is not used if the associated array has fewer than four dimensions, or has four non-variable dimensions.

Fifth Subscript Pointer Field: This field contains either a pointer to the dictionary entry for the sixth dimension factor, which has a value of D1*D2*D3*D4*D5*L, or a pointer to the dictionary entry for the fifth subscript parameter used to dimension the associated array if that array has variable dimensions. This field is not used if the associated array has fewer than five dimensions, or has five non-variable dimensions.

Sixth Subscript Pointer Field: This field contains either a pointer to the dictionary entry for the seventh dimension factor, which has a value of D1*D2*D3*D4*D5*D6*L, or a pointer to the dictionary entry for the sixth subscript parameter used to dimension the associated array if that array has variable dimensions. This field is not used if the associated array has fewer than six dimensions, or has six nonvariable dimensions.

Pointer to Last Subscript Parameter: This field contains a pointer to the dictionary entry for the seventh subscript parameter used to dimension the associated array if that array has variable dimensions. This field is not used if the associated array has fewer than seven dimensions, or has seven nonvariable dimensions.

COMMON Table

The COMMON table contains: (1) COMMON block name entries, which describe COMMON blocks; (2) equivalence group entries, which describe equivalence groups; and (3) equivalence variable entries, which describe equivalence variables.

COMMON BLOCK NAME ENTRY FORMAT: The format of the COMMON block name entries constructed by phase 10 is illustrated in Figure 27.

Chain Field: The chain field is used to maintain linkage between the various COMMON block name entries. It contains either a pointer to the next COMMON block name entry or an indicator (zero), which indicates that additional common blocks have not yet been encountered.

P1 Field: The P1 field contains a pointer to the dictionary entry for the first variable in this COMMON block.

P2 Field: The P2 field contains a pointer to the dictionary entry for the last variable in this COMMON block.

Name Field: The name field contains the name (right-justified) of the COMMON block for which this COMMON block name entry was constructed.

Character Number Field: The character number field contains the number of characters in the COMMON block name.

ISN Field: The ISN field contains the ISN assigned to the statement in which this COMMON block name first occurs.

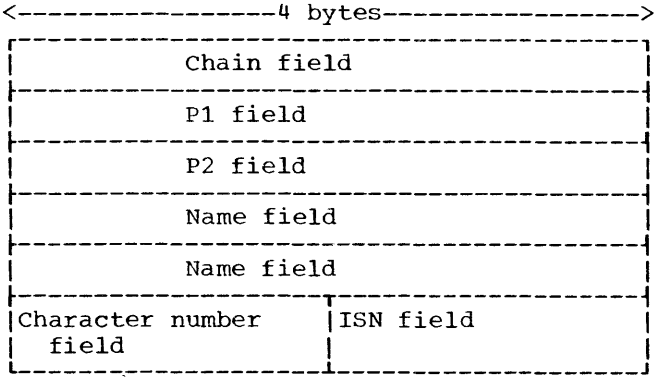


Figure 27. Format of a COMMON Block Name Entry

MODIFICATIONS TO COMMON BLOCK NAME ENTRIES: During compilation, certain fields of COMMON block name entries may be modified. Figure 28 illustrates the format of a COMMON block name entry after COMMON block processing by subroutine STALL-IEKGST and CORAL. Only changes are indicated; *stands for unchanged.

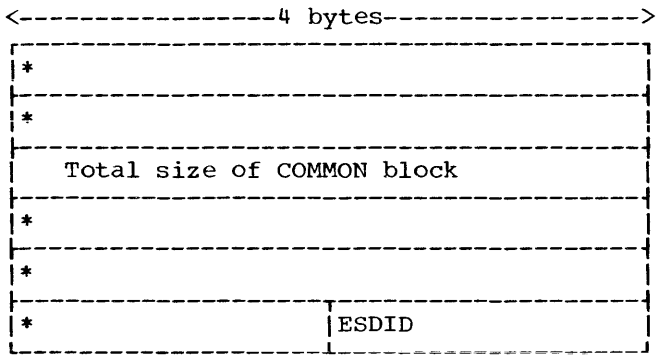


Figure 28. Format of COMMON Block Name Entry After COMMON Block Processing

EQUIVALENCE GROUP ENTRY FORMAT: The format of the equivalence group entries constructed by phase 10 is illustrated in Figure 29.

Indicator Field: The indicator field is nonzero if a variable in this group is subscripted and its DIMENSION statement has not been processed.

Chain Field: The chain field is used to maintain linkage between the various equivalence groups. It contains a pointer to the next equivalence group entry.

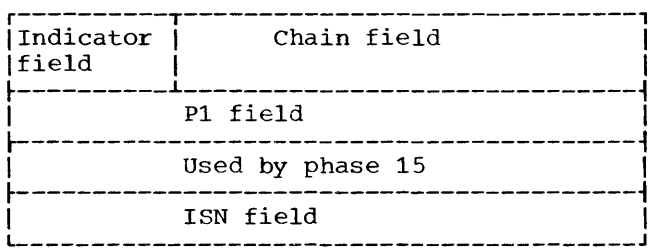


Figure 29. Format of an Equivalence Group Entry

P1 Field: The P1 field contains a pointer to the equivalence variable entry for the first variable in the equivalence group or for the first variable in the COMMON block.

ISN Field: The ISN field contains the ISN assigned to the statement in which any name of the EQUIVALENCE group first occurs.

MODIFICATIONS TO EQUIVALENCE GROUP ENTRIES: During compilation, certain fields of equivalence group entries may be modified. Figure 30 illustrates the format of an equivalence group entry after equivalence processing by subroutine STALL-IEKGST. Only changes are indicated; * stands for unchanged.

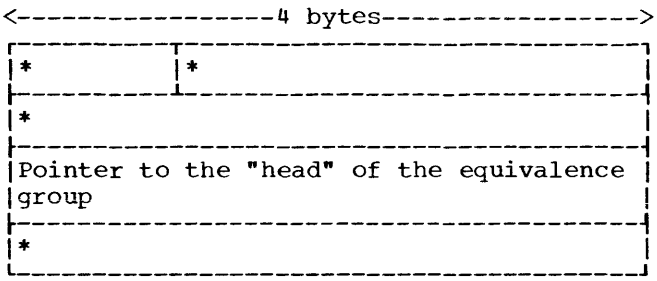


Figure 30. Format of Equivalence Group Entry After Equivalence Processing

EQUIVALENCE VARIABLE ENTRY FORMAT: The format of the equivalence variable entries constructed by phase 10 is illustrated in Figure 31.

Indicator Field: The indicator field is nonzero if the equivalence variable is subscripted prior to being dimensioned.

P1 Field: The P1 field contains a pointer to the dictionary entry for this equivalence variable.

Number of Subscripts Field: The number of subscripts field contains the total number of subscripts used by a variable being equivalenced, with subscripts, prior to being dimensioned.

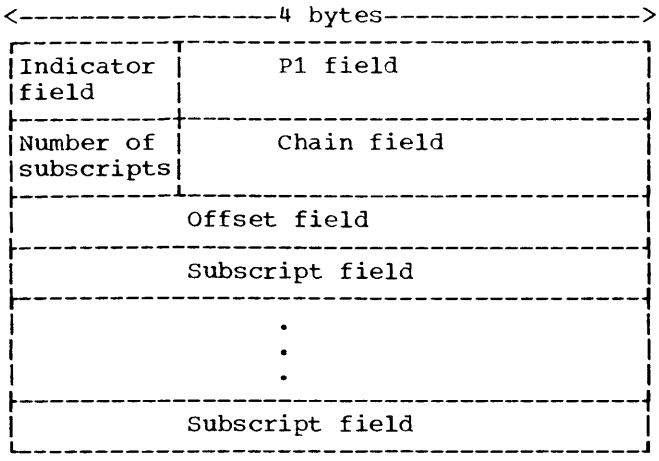


Figure 31. Format of Equivalence Variable Entry

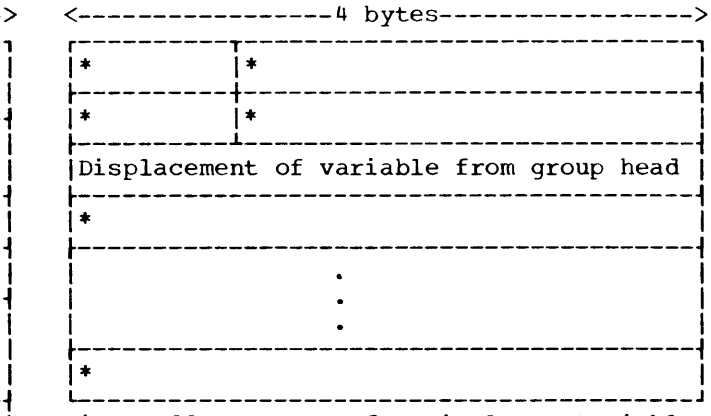


Figure 32. Format of Equivalence Variable Entry After Equivalence Processing

Chain Field: The chain field is used to maintain linkage between the various variables in the equivalence group. It contains a pointer to the equivalence variable entry for the next variable in the equivalence group.

Offset Field: The offset field contains the displacement of this variable from the first element in the equivalence group.

Subscript Field: The subscript field(s) contains the actual subscript(s) specified for a variable being equivalenced, with subscripts, prior to being dimensioned.

MODIFICATIONS TO EQUIVALENCE VARIABLE ENTRIES: During compilation, certain fields of equivalence variable entries may be modified. Figure 32 illustrates the format of an equivalence variable entry after equivalence processing by the STALL-IEKGST subroutine. Only changes are indicated; * stands for unchanged.

Literal Table

The literal table contains literal constant entries, which describe literal constants used as arguments in CALL statements, and literal data entries, which describe the literal data appearing in DATA statements. (Entries for literal data appearing in DATA statements are not chained. They are pointed to from data text.)

LITERAL CONSTANT ENTRY FORMAT: The format of the literal constant entries constructed by phase 10 is illustrated in Figure 33.

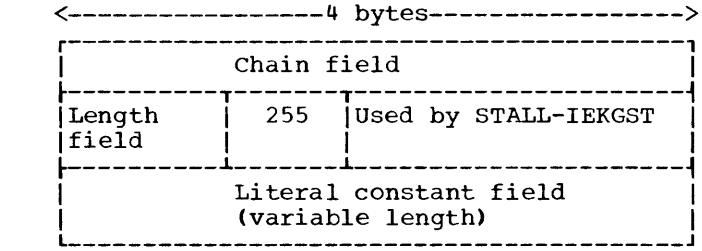


Figure 33. Format of Literal Constant Entry

Chain Field: The chain field is used to maintain linkage between the various literal constant entries. It contains a pointer to the previous literal constant entry.

Length Field: The length field contains the length (in bytes) of the literal constant.

Literal Constant Field: The literal constant field contains the actual literal constant for which the entry was constructed. The field ranges from 1 to 255 bytes (1 character/byte, left-justified) depending on the size of the literal constant.

MODIFICATIONS TO LITERAL CONSTANT ENTRIES: During compilation, certain fields of literal constant entries may be modified. Figure 34 illustrates the format of a literal constant entry after literal processing by STALL-IEKGST. Only changes are indicated; * stands for unchanged.

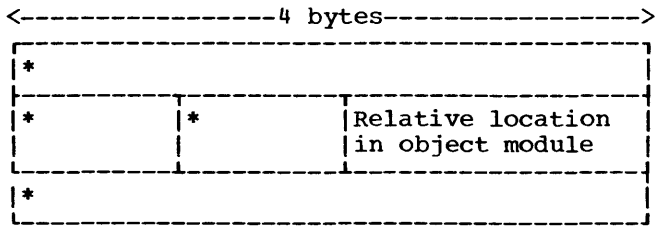


Figure 34. Format of Literal Constant Entry After Literal Processing

LITERAL DATA ENTRY FORMAT: The format of the literal data entries constructed by phase 10 is illustrated in Figure 35.

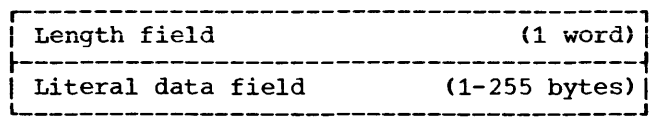


Figure 35. Format of Literal Data Entry

Length Field: The length field contains the length (in bytes) of the literal data for which the entry was constructed.

Literal Data Field: The literal data field contains the actual literal data. The field ranges from 1 to 255 bytes (1 character/byte, left-justified) depending on the size of the literal data.

Branch Tables

The branch tables contain initial branch table entries and standard branch table entries. An initial branch table entry is constructed by phase 10 as it encounters each computed GO TO statement of the source module. Standard branch table entries are constructed by phase 10 for each statement number appearing in the computed GO TO statement.

INITIAL BRANCH TABLE ENTRY FORMAT: The format of the initial branch table entries constructed by phase 10 is illustrated in Figure 36.

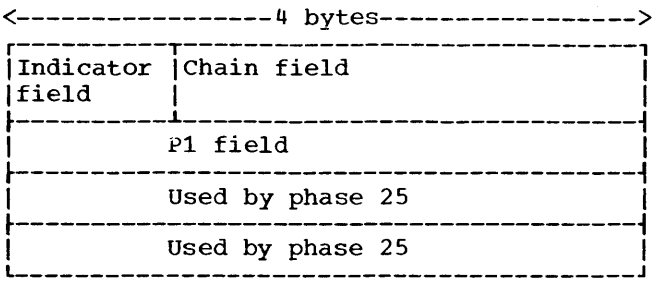


Figure 36. Format of Initial Branch Table Entry

Indicator Field: The indicator field is nonzero for an initial branch table entry. This indicates that the entry is for compiler-generated statement number for the "fall-through" statement. (The fall-through statement is executed if the value of the control variable is equal to zero or larger than the number of statement numbers in the computed GO TO statement.)

Chain Field: The chain field is used to maintain linkage between the various branch table entries. It contains a pointer to the next branch table entry.

P1 Field: The P1 field contains a pointer to the statement number/array table entry for the compiler-generated statement number for the fall-through statement.

MODIFICATIONS TO INITIAL BRANCH TABLE ENTRIES: During compilation, certain fields of initial branch table entries may be modified. Figure 37 illustrates the format of an initial branch table entry after phase 25 processing is complete. Only changes are indicated; * stands for unchanged.

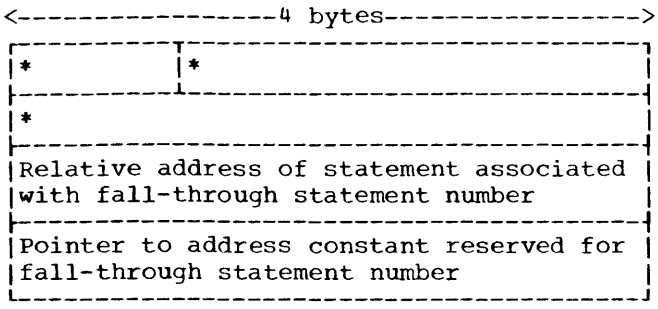


Figure 37. Format of Initial Branch Table Entry After Phase 25 Processing

STANDARD BRANCH TABLE ENTRY FORMAT: The format of the standard branch table entries constructed by phase 10 is the same as the format for initial branch table entries.

Indicator Field: This field is zero for standard branch table entries.

Chain Field: This field is used to maintain linkage between the various branch table entries. It contains a pointer to the next branch table entry.

P1 Field: The P1 field contains a pointer to the statement number/array table entry for the statement number (appearing in a computed GO TO statement) for which the standard branch table entry was constructed.

MODIFICATIONS TO STANDARD BRANCH TABLE ENTRIES: During compilation, certain fields of standard branch table entries may be modified. Figure 38 illustrates the format of a standard branch table entry after the processing of phase 25 is complete. Only changes are indicated; * stands for unchanged.

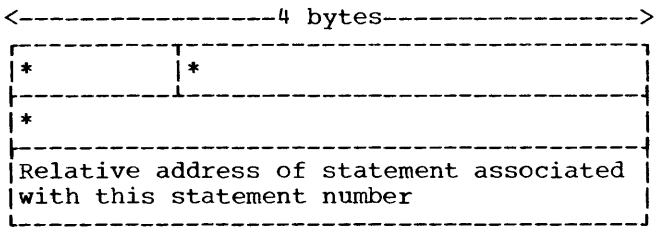


Figure 38. Format of Standard Branch Table Entry After Phase 25 Processing

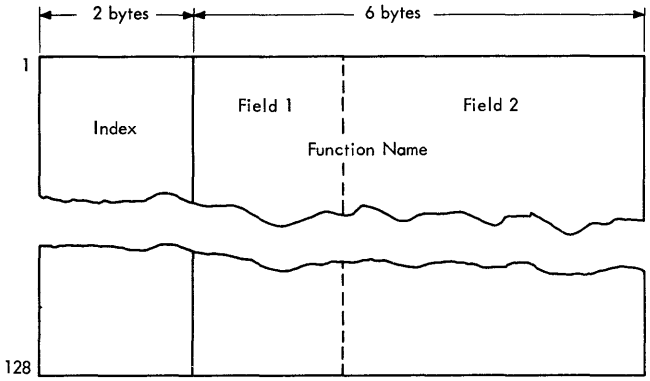
FUNCTION TABLE

The function table (IEKLFT) contains entries for the IBM supplied function subprograms and in-line routines. The subprograms reside on the FORTRAN system library (SYS1.FORTLIB), while the in-line routines are expanded at compile time. The function table is used by phase 15 to determine the validity of the arguments to the function subprogram.

Each entry in the function table (see Table 23) contains two fields: an index field (2 bytes) and a function name field (6 bytes).

Function Name Field: This field contains the names of all library and in-line functions. It is searched in ascending order beginning with field 1 and then with field 2. Field 1 contains the four low-order characters of the name; field two contains the two high-order characters of the name.

Table 23. Function Table -- IEKLFT (12, 128)



Index Field: This field contains a pointer to entries in the following tables:

FUNTB1(128) -- This table contains 128 1-byte entries pointing back to the function table.

FUNTB2(128) -- This table contains 128 1-byte entries which give the mode of the arguments for all library and in-line functions.

FUNTB3(128) -- This table contains 60 1-byte entries which give the mode of the result for all in-line functions. The first 68 bytes of the table are not used.

FUNTB4(68) -- This table contains 68 4-byte locations reserved for dictionary pointers to library routines.

TEXT OPTIMIZATION BIT TABLES

There are nine major bit tables used extensively throughout text optimization. These tables (each four words or 128 bits in length) contain bits that are preset. Only the first 86 bit positions in each table are meaningful and each of these is associated with a particular text entry operator. The settings (on or off) given to these bits indicate either the validity of operand positions in a text entry with a particular operator or the candidacy of a text entry with a particular operator for text optimization procedures.

Three of these tables, MVW, MVU, and MVV are tested by subroutine KORAN-IEKQKO and indicate the validity of the operand positions in a text entry with a given operator. The MVW table indicates the validity of the operand 1 position; the MVU table indicates the validity of the operand 2 position; and the MVV table indicates the validity of the operand 3 position. For example, if the bit in MVW that corresponds to a particular operator is set to on, then the operand 1 position of a text entry having that operator contains a valid or actual operand. If the bit is set to off, the operand 1 position of the text entry does not contain an actual operand. (In the latter case, the operand 1 position may still contain information that is pertinent to the text entry; however, it does not contain an actual operand.)

The remaining six tables, MBM, MSGM, MGM, MXM, MSM, and MBR are also tested by subroutine KORAN-IEKQKO and indicate the candidacy of a text entry with a particular operator for text optimization procedures. The MBM table indicates whether or not text entries with a particular operator are to be considered for backward movement; the MXM table indicates whether or not text entries with a particular operator are to be considered for common expression elimination; the MSM table indicates whether or not text entries with a particular operator are to be considered for strength reduction; and the MBR table indicates whether or not the operator is a branch.

The text optimization bit tables are illustrated in Table 24. In this table, the operator associated with each bit position in the bit tables is identified. The bits settings for each operator as they appear in the bit tables is also shown. An x signifies that the bit is on; a blank signifies that the bit is off.

Table 24. Text Optimization Bit Tables

Bit	Operator	Bit Tables								
		MVW	MVU	MVV	MSGM	MBM	MXM	MSM	MBR	MGM
1	•NOT•	X	X			X	X			
2	UNARY MINUS	X	X			X	X			
3										
4	•AND•	X	X	X		X	X			
5	)									
6	•OR•	X	X	X		X	X			
7	•XOR•	X	X	X	X	X	X			
8	ST	X	X			X				
9	, (ARG)	X	X	X					X	
10	+	X	X	X	X	X	X	X		X
11	-	X	X	X	X	X	X	X		X
12	*	X	X	X	X	X	X			X
13	/	X	X	X	X	X	X			X
14	LA	X	X	X		X				
15	EXT	X								
16	BG		X	X	X			X	X	
17	BL		X	X	X			X	X	
18	BNE		X	X					X	
19	BGE		X	X	X			X	X	
20	BLE		X	X	X			X	X	
21	BE		X	X					X	
22	SC	X	X	X	X	X	X			X
23	I/O LIST	X	X						X	
24	BCOMP			X					X	
25	(									
26	EM									
27	B									
28	BA		X						X	
29	BBT		X	X					X	
30	BBF		X	X					X	
31	LBIT	X	X			X	X		X	
32	BGZ		X						X	
33	BLZ		X						X	
34	BNEZ		X						X	
35	BGEZ		X						X	
36	BLEZ		X						X	
37	BEZ		X						X	
38										
39	NMLST	X	X							
40										
41	BF		X						X	
42	BT		X						X	
43	LDB	X		X		X				

Bit	Operator	Bit Tables								
		MVW	MVU	MVV	MSGM	MBM	MXM	MSM	MBR	MGM
44	LIBF	X				X	X			
45	RS	X	X		X	X	X			X
46	LS	X	X		X	X	X			X
47	BXHLE									
48										
49										
50	•LE•	X	X	X		X	X			
51	•GE•	X	X	X		X	X			
52	•EQ•	X	X	X		X	X			
53	•LT•	X	X	X		X	X			
54	•GT•	X	X	X		X	X			
55	•NE•	X	X	X		X	X			
56	MAX2	X	X	X		X	X			
57	MIN2	X	X	X		X	X			
58	DIM	X	X	X		X	X			
59	IDIM	X	X	X		X	X			
60	DMOD	X	X	X		X	X			
61	MOD	X	X	X		X	X			
62	AMOD	X	X	X		X	X			
63	DSIGN	X	X	X		X	X			
64	SIGN	X	X	X		X	X			
65	ISIGN	X	X	X		X	X			
66	DABS	X	X			X	X			
67	ABS	X	X			X	X			
68	IABS	X	X			X	X			
69	IDINT	X	X			X	X			
70										
71	INT	X	X			X	X			
72	HFIX	X	X			X	X			
73	IFIX	X	X			X	X			
74	DFLT	X	X			X	X			
75	FLT	X	X			X	X			
76	DBLE	X	X			X	X			
77	BITON	X	X							
78	BITOFF	X	X							
79	BITFLP	X	X							
80	ANDF	X	X	X		X	X			
81	ORF	X	X	X		X	X			
82	COMPL	X	X			X	X			
83	MOD24	X	X			X	X			
84	LCOMPL	X	X			X	X			
85	SHFTR	X	X	X		X	X			
86	SHFTL	X	X	X		X	X			

REGISTER ASSIGNMENT TABLES

The register assignment tables are a set of one-dimensional arrays used by the full register assignment routines of phase 20. There are three types of tables: local assignment tables (see Table 25), global assignment tables (see Table 27), and register usage tables. The register usage tables are work tables used by the local and global assignment routines in the process of full register assignment.

Register Use Table

The format of the register use tables, TRUSE and RUSE, are the same for the local and global assignment routines. Each table is 16 words long. Words 2 through 11 represent general registers 2 through 11; words 12, 14, and 16 represent floating-point point registers 2, 4, and 6; words 1, 13, and 15 are unused.

If the contents of TRUSE(i) and RUSE(i) is equal to zero, then register i is available for assignment. If the value contained in TRUSE(i) or RUSE(i) is between 2 and 128, inclusive, then the register i is assigned to the variable whose MCOORD value is equal to the contents of TRUSE(i) or RUSE(i). If the contents of TRUSE(i) or RUSE(i) has a value between 252 and 255, register i is unavailable for assignment and is reserved for special use (see next paragraph).

Register Use Considerations: Registers 15 and 14 are not available for use by register assignment. They are reserved and used for branching during the execution of the object module resulting from the compilation.

Table 25. Local Assignment Tables

Name	Function	Origin[1]
J	Serves as index to TXP, BVP, BVRA, BVA.	FWDPAS-IEKRFP
TXP	Gives the storage location of the text item associated with each value of J.	FWDPAS-IEKRFP
BVP	Contains the MCOORD value associated with operand 1 of the text item represented by J.	FWDPAS-IEKRFP
BVRA	Indicates the register locally assigned to the quantity represented by J.	BKPAS-IEKRBP
BVA	Represents the activity within the block of the quantity represented by J; also contains indicator bits describing the quantity (see Table 25).	FWDPAS-IEKRFP
WJ[2]	Indicates whether a variable is eligible for local assignment. Indexed via the MCOORD values obtained from BVP. Text item number of first definition = J.	FWDPAS-IEKRFP

[1]This column indicates the name of the register assignment routine that initially creates the particular table.
[2]Although WJ is distinctly a local assignment table, it is indexed by the quantity MCOORD (which is used to index the global assignment tables) rather than by the local assignment table index, J.

Table 26. BVA Table

Bit	Meaning
0	Not used.
1	Text item is candidate for forward movement.
2	P1 is a temporary used as an argument.
3	Inhibit 'inter-block' register assignment for text item.
4	Text item is candidate for 'inter-block' register assignment.
5	Text item is candidate for floating-point downgrading if a CALL statement is found.
6	Text item is candidate for register classification.
7	P1 is the result of an integer mod function.
8	The operand has been encountered before.
9	Text item is the imaginary result of a complex function.
10	The operand is defined by a function call.
11	P1 is floating-point.
12	One of the operands is the result of an integer multiply or divide.
13	Zero length temporary indicator.
14	Case II subscript indicator. Text item was changed to a Case II from Case I.
15 . . . 31	BVA - Local Activity.

Note: The BVA table consists of a fullword for each text in the block.

Table 27. Global Assignment Tables

Name	Function	Origin
MCOORD	Serves as an index to MVD, EMIN, RA, RAL, WABP, WA and WJ.	Phase 15
MVD	Gives the location of the dictionary entry for the variable associated with the given value of MCOORD.	Phase 15
EMIN	Indicates whether the variable associated with a particular MCOORD value is eligible for global assignment.	REGAS-IEKRRG
RA	Indicates the number of the first register globally assigned to the variable represented by the MCOORD value; provides continuity in global assignment from inner to outer loops.	GLOBAS-IEKRGB
RAL	Indicates the register globally assigned to the variable represented by the MCOORD value.	GLOBAS-IEKRGB
WA	Indicates the total activity for the variable represented by the MCOORD value. Calculated by adding 4. to the value each time a definition of the variable is encountered and adding 3. to the value for a use of the variable.	FWDPAS-IEKRFP
WABP	Indicates the activity of base variables. Calculated in the same manner as the WA table.	FWDPAS-IEKRFP

Register 13 is not available for use by register assignment. It is reserved and used during the execution of the object module to contain the address of the save area set aside for the object module (see "Generation of Initialization Instructions" under "Section 2: Discussion of Major Components" in this publication). Register 13 is also used to:

- Branch tables for computed GO TO statements
- Parameter list for external references
- Local constants, variables, and arrays
- Adcons for external references

If the above items exceed 4096 bytes, the adcons are referred to by register 12.

Register 12 is not available for use by register assignment.

Registers 11, 10, and 9 may or may not be available for use by register assignment. Their use depends upon the number of required reserved registers (see Phase 20, "Branching Optimization").

NAMELIST DICTIONARIES

Namelist dictionaries are developed by CORAL for the NAMELIST statements appearing in the source module. These dictionaries provide IHCNAMEL with the information required to implement READ/WRITE statements using NAMELIST statements. The namelist dictionary constructed by CORAL from the phase 10 namelist text representation of each NAMELIST statement contains an entry for the namelist name and entries for the variables and arrays associated with that name.

NAMELIST NAME ENTRY FORMAT: The format of the entry constructed for the namelist name is illustrated in Figure 39.

Figure 39. Format of Namelist Name Entry

Name Field: The name field contains the namelist name, right-justified, with leading blanks.

NAMELIST VARIABLE ENTRY FORMAT: The format of the entry constructed for a variable appearing in a NAMELIST statement is illustrated in Figure 40.

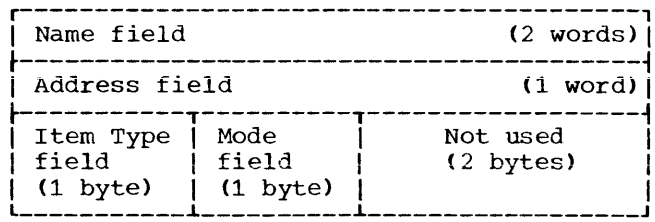


Figure 40. Format of Namelist Variable Entry

Name Field: The name field contains the name of the variable, right-justified, with leading blanks.

Address Field: The address field contains the relative address of the variable.

Item Type Field: This field is zero for a variable.

Mode Field: The mode field contains the mode of the variable.

NAMELIST ARRAY ENTRY FORMAT: The format of the entry constructed for an array appearing in a NAMELIST statement is illustrated in Figure 41.

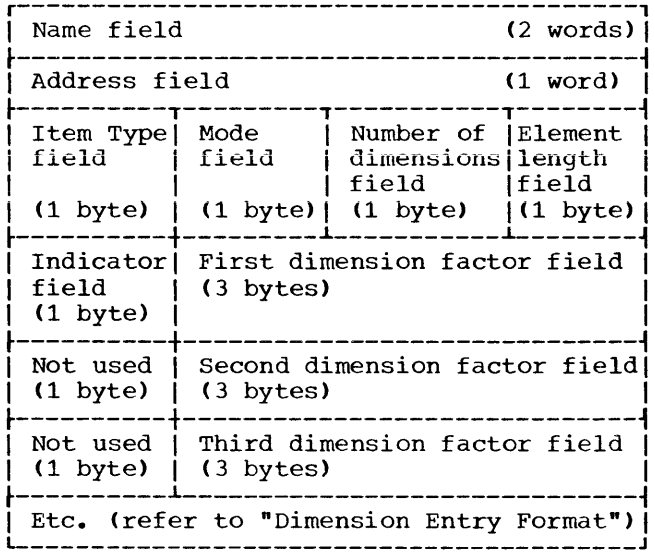


Figure 41. Format of Namelist Array Entry

Name Field: The name field contains the name of the array, right-justified, with leading blanks.

Address Field: The address field contains the relative address of the beginning of the array.

Item Type Field: This field is nonzero for an array.

Mode Field: This field contains the mode of the elements of the array.

Number of Dimensions Field: This field contains the number of dimensions (1 through 7) of the associated array.

Element Length Field: The element length field contains the length of each element in the associated array.

Indicator Field: This field is zero if the associated array has variable dimensions; otherwise, it is nonzero.

First Dimension Factor Field: If the associated array does not have variable dimensions, this field contains the total size of the array. If the array has variable dimensions, this field contains the relative address of first subscript parameter used to dimension the array.

Second Dimension Factor Field: If the associated array does not have variable dimensions, this field contains the location of the second dimension factor (D1*L). If the array has variable dimensions, this field contains the relative address of the second subscript parameter used to dimension the array.

Third Dimension Factor Field: If the associated array does not have variable dimensions, this field contains the location of the third dimension factor (D1*D2*L). If the array has variable dimensions, this field contains the relative address of the third subscript parameter used to dimension the array.

DIAGNOSTIC MESSAGE TABLES

There are two major diagnostic tables associated with error message processing by phase 30: the error table and the message pointer table.

ERROR TABLE

The error table is constructed by phases 10 and 15. As source statement errors are encountered by these phases, corresponding entries are made in the error table. Each error table entry consists of 2 one-word fields. The first field contains the message number associated with the particular error. The message numbers that can appear in the error table are those associated with messages of error code levels 4 and 8 (refer to the publication IBM System/360 Operating System: FORTRAN IV (G and H) Programmer's Guide). The second field contains either an internal statement number, if the entry is for a statement that is in error, a dictionary pointer, if the entry is for a symbol that is in error (e.g., a variable that is incorrectly used in an EQUIVALENCE statement), or a statement number, if the entry is for an undefined statement number.

MESSAGE POINTER TABLE

The message pointer table contains an entry for each message number that may appear in an error table entry. Each entry in the message pointer table consists of a single word. The high-order byte of the word contains the length of the message associated with the message number. The three low-order bytes contain a pointer to the text for the message associated with the message number.

Intermediate text is an internal representation of the source module from which the machine instructions of the object module are generated. The conversion from intermediate text to machine instructions requires information about variables, constants, arrays, statement numbers, in-line functions, and subscripts. This information, derived from the source statements, is contained in the information table, and is referred to by the intermediate text. The information table supplements the intermediate text in the generation of machine instructions by phase 25.

PHASE 10 INTERMEDIATE TEXT

Phase 10 creates intermediate text (in operator-operand pair format) for use as input to subsequent phases of the compiler. There are six types of intermediate text produced by phase 10:

- Normal text -- the operator-operand pair representations of source statements other than DATA, NAMELIST, DEFINE FILE, FORMAT, and Statement Functions (SF).
- Data text -- the operator - operand pair representations of DATA statements and the initialization constants in explicit type statements.
- Namelist text -- the operator-operand pair representations of NAMELIST statements.
- Define file text -- the operator-operand pair representation of DEFINE FILE statements.
- SF skeleton text -- the operator-operand pair representations of statement functions using sequence numbers as operands of the intermediate text entries. The sequence numbers replace the dummy arguments of the statement functions. This type of text is, in effect, a "skeleton" macro.
- Format text -- the internal representations of FORMAT statements.

Note: Intermediate text representations are, for subblock allocation, divided into only two main types: special (DATA, NAMELIST, DEFINE FILE, FORMAT, and SF skeleton text), and normal (text other than special text). The intermediate text representations are comprised of individual text entries. Each intermediate main text type is allocated unique subblocks of main storage. The subblocks that constitute an intermediate text area are obtained by phase 10, as needed, via requests to the FSD (see "Storage Distribution" under "FORTRAN System Director").

Intermediate Text Chains

Each intermediate text area (i.e., the subblocks allocated to a particular type of text) is arranged as a chain that links together (1) the text entries that are developed and placed into that area, and (2) in some cases, the intermediate text representation for individual statements.

The normal text chain is a linear chain of normal text entries; that is, each normal text entry is pointed to by the previously developed normal text entry.

The data text chain is bi-linear. This means that:

1. The text entries that constitute the intermediate text representation of a DATA statement are linked by means of pointers. Each text entry for the statement is pointed to by the previously developed text entry for the statement.
2. The intermediate text representations of individual DATA statements are linked by means of pointers, each representation being pointed to by the previously developed representation. (A special chain address field within the first text entry developed for each DATA statement is reserved for this purpose.)

The namelist text chain operates in the same manner as the data text chain.

The define file text chain is a linear chain of define file text entries, each define file text entry is pointed to by a previously developed define file text entry. A zero chain signals the end of all define file text for a program.

The SF skeleton text chain is linear only in that each text entry developed for

an operator-operand pair within a particular statement function is pointed to by the previous text entry developed for that same statement function. The intermediate text representations for separate statement functions are not chained together. However, a skeleton can readily be obtained by means of the pointer contained in the dictionary entry for the name of the statement function.

The format text chain consists of linkages between the individual intermediate text representations of FORMAT statements. The pointer field of the second text entry in the intermediate representation of a FORMAT statement points to the intermediate text representation of the next FORMAT statement. (The individual text entries that make up the intermediate text representation of a FORMAT statement are not chained.)

Format of Intermediate Text Entry

Those statements that undergo conversion from source representation to intermediate text representation are divided into operator-operand pairs, or text entries. Figure 42 illustrates the format of an intermediate text entry constructed by phase 10.

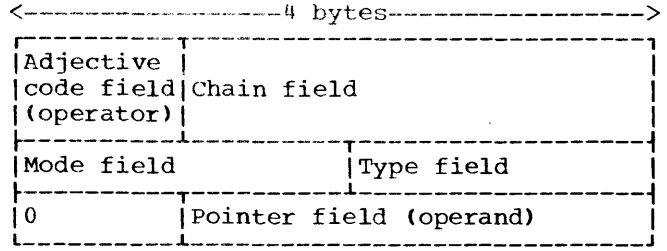


Figure 42. Intermediate Text Entry Format

Adjective Code Field: The adjective code field corresponds to the operator of the operator-operand pair. Operators are not entered into text entries in source form; they are converted to a numeric value as specified in the adjective code table (see Table 28). It is the numeric representation of the source operator that actually is inserted into the text entry. Primary adjective codes (operators that define the nature of source statements) also have numeric values.

Chain Field: The chain field is used to maintain linkage between intermediate text entries. It contains a pointer to the next text entry.

Mode and Type Fields: The mode and type fields contain the mode and type of the operand of the text entry. Both items appear as numeric quantities in a text entry and are obtained from the mode and type table (see Tables 21 and 22).

Pointer Field: The pointer field contains a pointer to the information table entry for the operand of the operator-operand pair. However, if the operand is a dummy argument of a statement function, the pointer field contains a sequence number, which indicates the relative position of the argument in the argument list.

Note: The text entries for FORMAT statements are not formatted as described in the foregoing. FORMAT text entries consist of the characters of the FORMAT statement in source format packed into successive text entries.

Table 28. Adjective Codes (Part 1 of 3)

Code (in decimal)	Mnemonic (where applicable)	Meaning
1	.NOT.	NOT
4	.AND.	AND
5	)	Right arithmetic parenthesis
6	.OR.	OR
7	.XOR.	Exclusive OR
8	=	Equal sign
9	,	Comma
10	+	Plus
11	-	Minus
12	*	Multiply
13	/	Divide
14	**	Exponentiation
15	(f	Function parenthesis
16	.LE.	Less than or equal
17	.GE.	Greater than or equal
18	.EQ.	Equal
19	.LT.	Less than

Table 28. Adjective Codes (Part 2 of 3)

Code (in decimal)	Mnemonic (where applicable)	Meaning
20	.GT.	Greater than
21	.NE.	Not equal
22	(s	Left subscript parenthesis
25	(	Left arithmetic parenthesis
26		End mark
71		GO TO, and implied branches
193		BLOCK DATA
205		DATA
208		SUBROUTINE, FUNCTION, or ENTRY
209		FORMAT (text)
210		End of I/O list
211		CONTINUE
212		Relative record number
213		Object time format variable
214		BACKSPACE
215		REWIND
216		END FILE
217		WRITE unformatted
218		READ unformatted
219		WRITE formatted
220		READ formatted
221		Beginning of I/O list
222	LDF	Statement number definition

Table 28. Adjective Codes (Part 3 of 3)

Code (in decimal)	Mnemonic (where applicable)	Meaning
223	GLDF	Generated statement number definition
225		WRITE using NAMELIST
226		READ using NAMELIST
227		FIND
230		I/O end-of-file parameter
231		I/O error parameter
232		BLANK
233	RET	RETURN
234	STOP	STOP
235		PAUSE
238		ASSIGN
240		Beginning of DO
241		Arithmetic assignment statement
242	NDOIF	End of DO 'IF'
243		Arithmetic IF
244		Relational IF
246		CALL
247	LIST	I/O or NAMELIST list item
248		NAMELIST
249	END	END
250		Computed GO TO
251		I/O unit number
252		FORMAT (statement numbers)
253		NAMELIST name

Examples of Phase 10 Intermediate Text

An example of each type of phase 10 text (normal, data, namelist, define file format, and SF skeleton) is presented below. For each type, a source language statement is first given. This is followed by the phase 10 text representation of that statement.

The phase 10 normal text representation of the arithmetic statement

```
100 A = B + C * D / E
```

is illustrated in Figure 43.

Adjective Code	Chain	Mode	Type	0	Pointer
Statement number definition		Statement number	0		→ 100
Arithmetic		Real	Scalar[1]		→ A
=		Real	Scalar[1]		→ B
+		Real	Scalar[1]		→ C
*		Real	Scalar[1]		→ D
/		Real	Scalar[1]		→ E
End mark[2]	To next normal text entry	0	0		ISN[3]
1 byte	3 bytes	2 bytes	2 bytes	1 byte	3 bytes

[1]Nonsubscripted variable.
[2]Operator of the special text entry that signals the end of the text representation of a source statement.
[3]Compiler generated sequence number used to identify each source statement.

Figure 43. Phase 10 Normal Text

The phase 10 data_text representation of the DATA statement

```
DATA A,B/2.1,3HABC/,C,D/1.,1./
```

is illustrated in Figure 44.

Adjective Code	Chain	Mode	Type	0	Pointer
DATA		0	ISN		→ To text for next DATA statement
0		Real	Scalar		→ A
,		Real	Scalar		→ B
/		Real	Constant		→ 2.1
,		Literal	Constant		→ 3HABC
/		Real	Scalar		→ C
,		Real	Scalar		→ D
/		Real	Constant		→ 1.
,	0	Real	Constant		→ 1.
1 byte	3 bytes	2 bytes	2 bytes	1 byte	3 bytes

Figure 44. Phase 10 Data Text

The phase 10 namelist text representation of the NAMELIST statement

```
NAMELIST /NAME1/A,B,C/NAME2/D,E,F/NAME3/G
```

where A and F are arrays is illustrated in Figure 45.

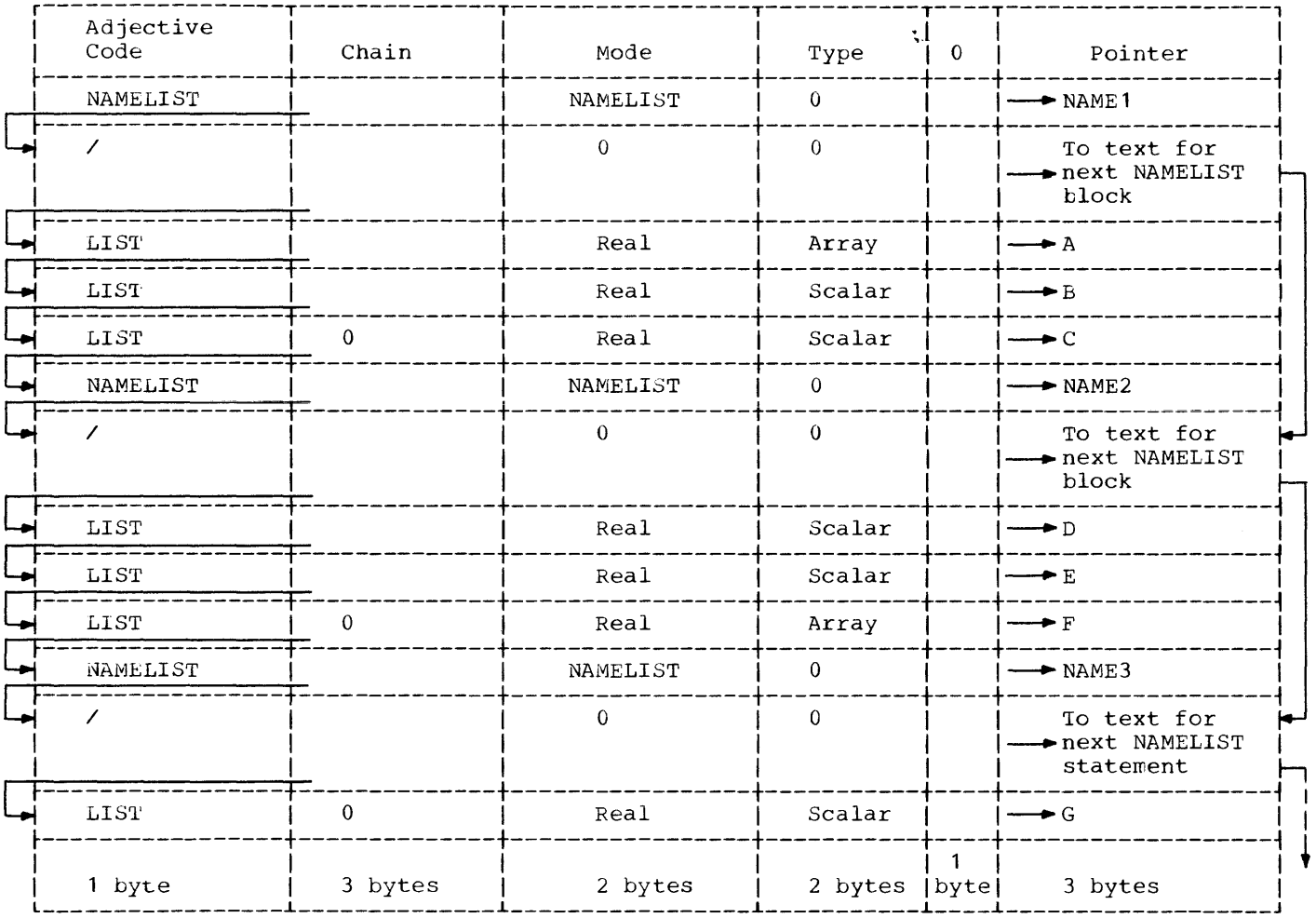


Figure 45. Phase 10 Namelist Text

The phase 10 define file text representation of the DEFINE FILE statement

DEFINE FILE $a_1(m_1,r_1,f_1,v_1)$

where a_1 is the input/output unit number, m_1 is the number of records, r_1 is the maximum record length, f_1 is the format code, and v_1 is the associated variable, is illustrated in Figure 46.

Adjective Code	Chain	Mode	Type	0	Pointer
I/O unit number		Integer	Constant		→ a_1
,		Integer	Constant		→ m_1
,		Integer	Constant		→ r_1
format code (f_1)	pointer to next define file text entry	Integer	Scalar		→ v_1
1 byte	3 bytes	2 bytes	2 bytes	1 byte	3 bytes

Figure 46. Phase 10 Define File Text

The phase 10 SF skeleton text representation of the statement function

ASF (A,B,C) = A+D*B*E/C

is illustrated in Figure 47.

Adjective Code	Chain	Mode	Type	0	Pointer
(		0	0		1
+		Real	Scalar		→D
*		0	0		2
*		Real	Scalar		→E
/		0	0		3
)		0	0		
End mark	0	0	0		0
1 byte	3 bytes	2 bytes	2 bytes	1 byte	3 bytes

Figure 47. Phase 10 SF Skeleton Text

The phase 10 format text representation of the FORMAT statement

```
5 FORMAT (2H0A,A6//5X,3
  (I4,E12.5,3F12.3,'ABC'))
```

is illustrated in Figure 48.

Pointer Code	Chain	Mode	Type	0	Pointer
Statement number definition		11	0		Statement number 5
FORMAT		ISN	0		To text for next FORMAT statement
. . .	. . .	. . .	. . .	. . .	. . .
(	2H0	A,	A6	/	/5X
,	3(I	4,	E1	2	.5,
3	F12	.3	,'	A	BC'
)	)≠ɓ [1]	ɓɓ	ɓɓ	ɓ	ɓɓɓ
1 byte	3 bytes	2 bytes	2 bytes	1 byte	3 bytes

[1]Group mark ('4F' in hexidecimal)

Figure 48. Phase 10 Format Text

PHASE 15/PHASE 20 INTERMEDIATE TEXT MODIFICATIONS

During phase 15 and phase 20 text processing, the intermediate text entries are modified to a format more suitable for optimization and object-code generation. The intermediate text modifications made by each phase are discussed separately in the following paragraphs.

PHASE 15 INTERMEDIATE TEXT MODIFICATIONS

The intermediate text input to phase 15 is the intermediate text created by phase 10. The intermediate text output of phase 15 is an expanded version of phase 10 intermediate text. The intermediate text output of phase 15 is divided into four categories:

- Unchanged text
- Phase 15 data text
- Statement number text
- Standard text

Unchanged Text

The unchanged text is the phase 10 normal text that is not changed but rearranged in format by phase 15 (see Figure 42). Unchanged text is passed on to subsequent phases with these modifications:

1. The mode and type fields are each expanded to a fullword.
2. A new word is inserted between the chain field and the mode field.
3. The adjective code is moved from the first byte of the chain field to the third byte of this new word.

Phase 15 Data Text

To facilitate the assignment of initial data values to their associated variables, phase 15 converts the phase 10 data text for DATA statements to phase 15 data text, which is in variable-constant format. The format of the phase 15 data text entries is illustrated in Figure 49.

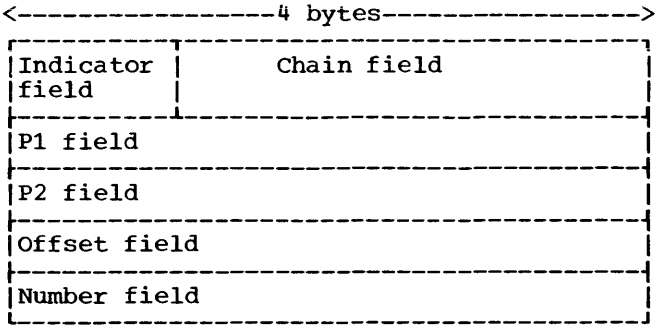


Figure 49. Format of Phase 15 Data Text Entry

Subfield	Function
Bit 0	not used
Bit 1	not used
Bit 2	not used
Bit 3	not used
Bit 4 'on'	initial data value is negative constant
Bit 5 'on'	initial data value is a literal constant
Bit 6 'on'	initial data value is in hexadecimal form
Bit 7 'on'	data table entry is six words long (variable is an array element).

Figure 50. Function of Each Subfield in Indicator Field of Phase 15 Data Text Entry

Indicator Field: The indicator field indicates the characteristics of the initial data value (constant) to be assigned to the associated variable. This field is one byte in length. The indicator field is divided into eight subfields, each of which is one bit long. The bits are numbered from 0 through 7. Figure 50 indicates the function of each subfield in the indicator field.

Chain Field: The chain field is used to maintain linkage between the various phase 15 data text entries. It contains a pointer to the next such entry.

P1 Field: The P1 field contains a pointer to the dictionary entry for the variable to which the initial data value is to be assigned.

P2 Field: The P2 field contains a pointer to the dictionary entry for the initial data value (constant) which is to be assigned to the associated variable.

Offset Field: The offset field contains the displacement of the subscripted variable from the first element in the array containing that variable. If the variable to which the initial data value is to be assigned is not subscripted, this field does not exist.

Number Field: The number field contains an indication of the number of successive items to which the initial data value is to be assigned. If the initial data value is not to be assigned to more than one item, this field does not exist.

Statement Number Text

The statement number text is an expanded version of the phase 10 intermediate text created for statement numbers. It is expanded to provide additional fields in which statistical information about the text block associated with the statement number is stored. The format of statement number text entries is illustrated in Figure 51.

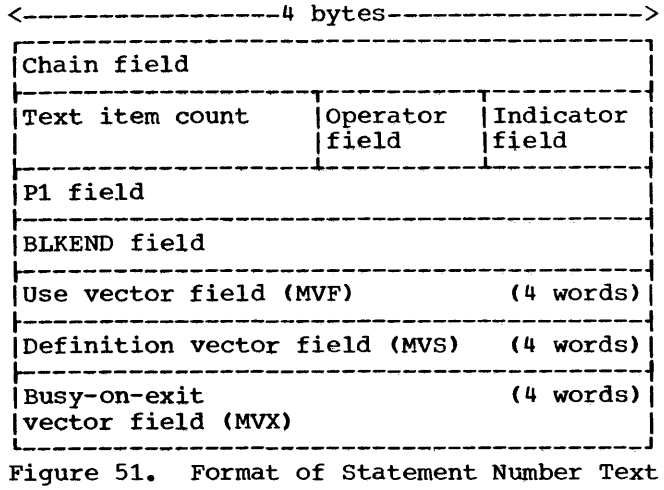


Figure 51. Format of Statement Number Text Entry

Chain Field: The chain field is used to maintain the linkage between the various intermediate text entries. It contains a pointer to the next text entry.

Text Item Count: The text item count is the total number of text items in the block, including the statement number text item itself and any end marks.

Operator Field: The operator field contains an internal operation code (numeric) for a statement number definition (see Table 29).

Indicator Field (ABFN): The indicator field is one byte long. This field indicates some of the characteristics of the text entries in the associated block. The indicator field contains eight subfields, each of which is one bit long. The subfields are numbered 0 through 7. Figure 52 indicates the function of each subfield in the indicator field.

Subfield	Function
Bits 0-3	not used
Bit 4 'on'	associated block contains an input/output operation
Bit 5 'on'	associated block contains a reference to a library function
Bit 6	not used
Bit 7 'on'	associated block contains an abnormal function reference

Figure 52. Function of Each Subfield in Indicator Field of Statement Number Text Entry

P1 Field: The P1 field contains a pointer to the statement number/array table entry for the statement number.

BLKEND Field: The BLKEND field contains a pointer to the last intermediate text entry within the block.

Use Vector Field (MVF): The use vector field is used to indicate which variables and constants are used in the associated block. Variables and constants, as they are encountered in the module by subroutine STALL-IEKGST are assigned a unique

co-ordinate (1 bit) in this vector field. In general, if the ith bit is set to on (1), the variable or constant assigned to the ith coordinate is used in the associated block. This field is used for OPT=1,2 only.

Definition Vector Field (MVS): The definition vector field is used to indicate which variables are defined in a block. Variables and constants, as they are encountered by subroutine STALL-IEKGST are assigned a unique coordinate (1 bit) in this vector field. In general, if the ith bit is set to on (1), the variable assigned to the ith coordinate is defined in the associated block. This field is used for OPT=1,2 only.

Busy-On-Exit Vector Field (MVX): The busy-on-exit vector field in phase 15 indicates which variables are not first used and then defined within the text block (not busy-on-entry). This field is converted by phase 20 to busy-on-exit data, which identifies those operands that are busy-on-exit from the block. Variables and constants, as they are encountered by subroutine STALL-IEKGST are assigned a unique coordinate (1 bit) in this vector field. In general, during phase 15, if the ith bit is set to on (1), the variable assigned to the coordinate is not busy-on-entry to the block. During phase 20, if the ith bit is set to on, the variable or constant assigned to the ith coordinate is busy-on-exit from the block. This field is used for OPT=2 only.

Table 29. Phase 15/20 Operators (Part 1 of 5)

Code (in decimal)	Mnemonic (where applicable)	Meaning
1	.NOT.	NOT
2	U	Unary minus
4	.AND.	.AND., LAND in-line routine
5	)	Right parenthesis
6	.OR.	.OR., LOR in-line routine
7	.XOR.	.XOR., LXOR in-line routine
8	ST	Load/Store
9	,	Argument
10	+	Plus
11	-	Minus
12	*	Multiply
13	/	Divide
14	LA	Load address
15	EXT	External function or subroutine CALL
16	BG	Branch greater than
17	BL	Branch less than
18	BNE	Branch not equal
19	BGE	Branch greater than or equal
20	BLE	Branch less than or equal
21	BE	Branch equal
22	SUB	Subscript
23	LIST	I/O list
24	BC	Branch computed
25	(	Left parenthesis
26	EM	End mark
27	B	Branch
28	BA	Branch assigned

Table 29. Phase 15/20 Operators (Part 2 of 5)

Code (in decimal)	Mnemonic (where applicable)	Meaning
29	BBT	Branch bit true
30	BBF	Branch bit false
31	LBIT	Logical value of bit
32	BGZ	Branch greater than zero
33	BLZ	Branch less than zero
34	BNEZ	Branch not equal to zero
35	BGEZ	Branch greater than or equal to zero
36	BLEZ	Branch less than or equal to zero
37	BEZ	Branch equal to zero
39	NMLS	NAMELIST operands (phase 20 only)
41	BF	Branch false
42	BT	Branch true
43	LDB	Load byte
44	LIBF	Library function call
45	RS	Right shift
46	LS	Left shift
47	BXHLE	Branch on index
48	ASSIGN	Assign
50	LE	Less than or equal
51	GE	Greater than or equal
52	EQ	Equal
53	LT	Less than
54	GT	Greater than
55	NE	Not equal
56	MAX2	MAX2 in-line routine
57	MIN2	MIN2 in-line routine

Table 29. Phase 15/20 Operators (Part 3 of 5)

Code (in decimal)	Mnemonic (where applicable)	Meaning
58	DIM	DIM in-line routine
59	IDIM	IDIM in-line routine
60	DMOD	DMOD in-line routine
61	MOD	MOD in-line routine
62	AMOD	AMOD in-line routine
63	DSIGN	DSIGN in-line routine
64	SIGN	SIGN in-line routine
65	ISIGN	ISIGN in-line routine
66	DABS	DABS in-line routine
67	ABS	ABS in-line routine
68	IABS	IABS in-line routine
69	IDINT	IDINT in-line routine
71	INT	INT in-line routine
72	HFIX	HFIX in-line routine
73	IFIX	IFIX in-line routine
74	DFLOAT	DFLOAT in-line routine
75	FLOAT	FLOAT in-line routine
76	DBLE	DBLE in-line routine
77	BITON	BITON in-line routine
78	BITOFF	BITOFF in-line routine
79	BITFLP	BITFLP in-line routine
80	AND	AND in-line routine
81	OR	OR in-line routine
82	COMPL	COMPL in-line routine
83	MOD24	MOD24 in-line routine

Table 29. Phase 15/20 Operators (Part 4 of 5)

Code (in decimal)	Mnemonic (where applicable)	Meaning
84	LCOMPL	LCOMPL in-line routine
85	SHFTR	SHFTR in-line routine
86	SHFTL	SHFTL in-line routine
100	LR	Load register (phase 20 only)
101	RC	Restore main storage (phase 20 only)
102	RR	Restore register (phase 20 only)
103		Register usage (phase 20 only)
104		STORE (phase 20 only) R13 as operand 2
203		Register usage (phase 20 only)
208		FUNCTION or SUBROUTINE or ENTRY
210		END input/output list
211		CONTINUE
212		Relative record number
213		Variable FORMAT
214		BACKSPACE
215		REWIND
216		END FILE
217		WRITE unformatted

Table 29. Phase 15/20 Operator (Part 5 of 5)

Code (in decimal)	Mnemonic (where applicable)	Meaning
218		READ unformatted
219		WRITE formatted
220		READ formatted
221		Begin input/output list
222	LDF	Statement number definition
223	GLDF	Generated statement number definition
225		WRITE using NAMELIST
226		READ using NAMELIST
227		FIND
230		Input/output end-of-file parameter
231		Input/output error parameter
233	RET	RETURN
234	STOP	STOP
235		PAUSE
249	END	END
251		Input/output unit number
252		FORMAT statement number
253		NAMELIST name

Standard Text

The standard text is an expanded and modified form of phase 10 intermediate text that is more suitable for optimization. The format of standard text entries is illustrated in Figure 53.

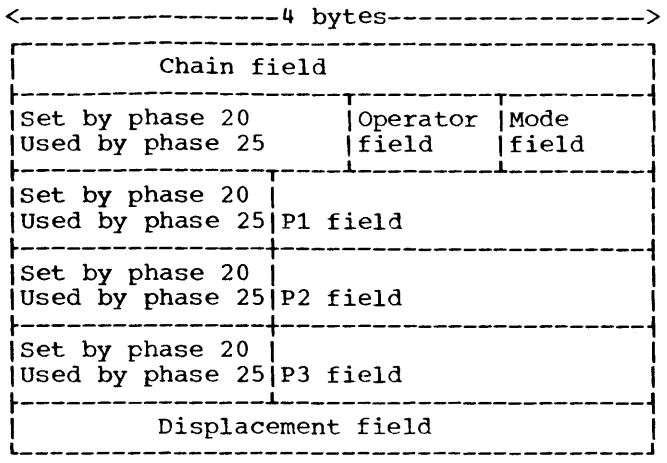


Figure 53. Format of a Standard Text Entry

Chain Field: The chain field is used to maintain the linkage between the various intermediate text entries. It contains a pointer to the next text entry.

Operator Field: The operator field contains an internal operation code (numeric) that indicates either the nature of the statement or the operation to be performed (see Table 29).

P1 Field: The P1 field contains either a pointer to the dictionary entry or statement number/array table entry for operand 1 of the text entry, or zero (0) if operand 1 does not exist.

P2 Field: The P2 field contains either a pointer to the dictionary entry for operand 2 of the text entry or zero (0) if operand 2 does not exist.

P3 Field: The P3 field contains either a pointer to the dictionary entry for operand 3 of the text entry, a pointer to a parameter list in the adcon table, an actual constant (for shifting operations), or zero (0) if operand 3 does not exist.

Mode Field: The mode field indicates the general mode of the expression and the mode of the operands. The bits are set by phase 15. The mode field can be referred to only as the fourth byte of the status mode word, which consists of a status field (2 bytes), an operator field (1 byte), and the mode field (1 byte). The status portion of the status mode word is explained later under "Phase 20 Intermediate Text Modification." The meanings of the bits in the mode field are given in Table 30.

Displacement Field: The displacement field appears only for subscript and load address text entries; it contains a constant displacement (if any) computed from constants in the subscript expression.

PHASE 20 INTERMEDIATE TEXT MODIFICATION

The intermediate text input to phase 20 is the output text from phase 15. The intermediate text output of phase 20 is of the same format as the standard text output of phase 15. The format of the phase 20 output text is illustrated in Figure 54.

R1, R2, and R3 Fields: The R1, R2, and R3 fields (each 4 bits long) are filled in by phase 20 during register assignment, and are referred to by phase 25 during the code generation process. The assigned registers are the operational registers for operand 1, operand 2, and operand 3, respectively.

Table 30. Meanings of Bits in Mode Field of Standard Text Entry Status Mode Word

Mode	Bits	Meaning
general	26	1 - indicates to phase 20 that this text entry is part of a subscript computation.
general	27-28	00 - LOGICAL 01 - INTEGER 10 - REAL or COMPLEX
operand 1	29	0 - short mode (LOGICAL*1, INTEGER*2, REAL*4, COMPLEX*8) 1 - long mode (LOGICAL*4, INTEGER*4, REAL*8, COMPLEX*16)
operand 2	30	0 - short mode (LOGICAL*1, INTEGER*2, REAL*4, COMPLEX*8) 1 - long mode (LOGICAL*4, INTEGER*4, REAL*8, COMPLEX*16)
operand 3	31	0 - short mode (LOGICAL*1, INTEGER*2, REAL*4, COMPLEX*8) 1 - long mode (LOGICAL*4, INTEGER*4, REAL*8, COMPLEX*16)

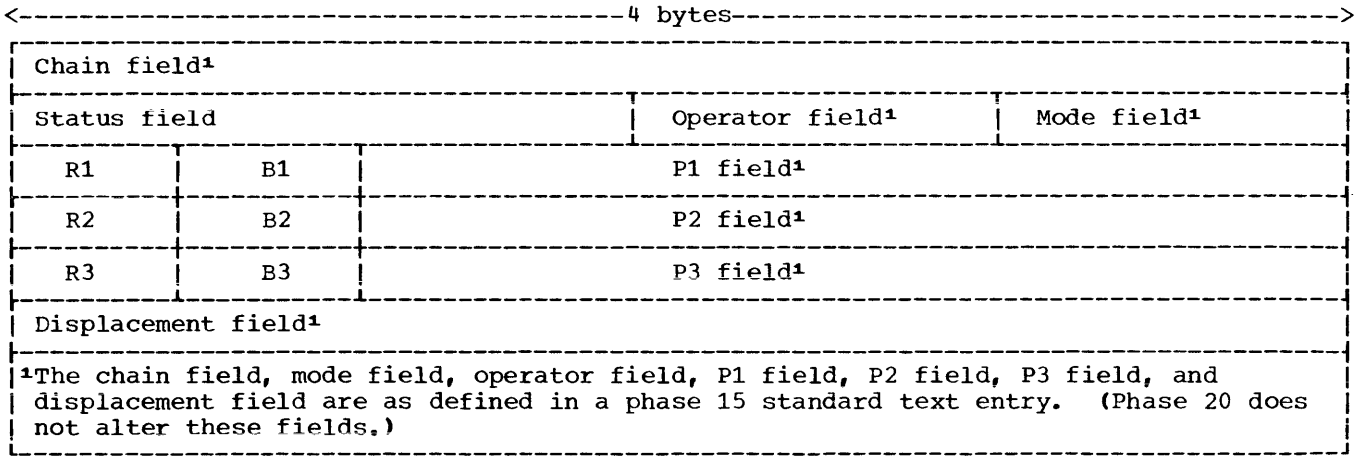


Figure 54. Format of Phase 20 Text Entry

B1, B2, and B3 Fields: The B1, B2, and B3 fields (each 4 bits long) are filled in by phase 20 during register assignment, and are referred to by phase 25 during the code generation process. The assigned registers are the base registers for operand 1, operand 2, and operand 3, respectively.

Status Field: The status field, the first two bytes of the status mode word, is set by phase 20 to indicate the status of the operands and the status of the base addresses of the operands in a text entry. The information in the status field is used by phase 25 to determine the machine instructions that are to be generated for the text entry. The status field bits and their meanings are illustrated in Table 31.

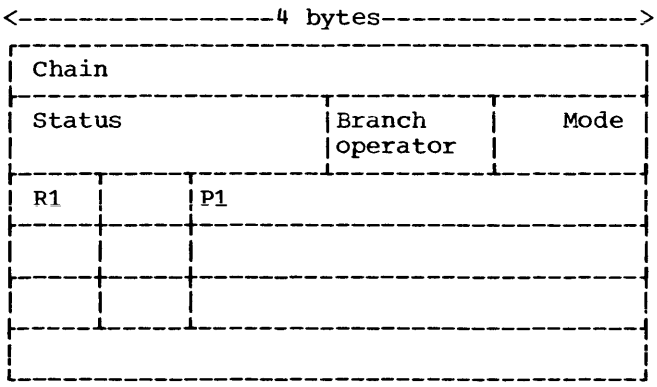
STANDARD TEXT FORMATS RESULTING FROM PHASES 15 AND 20 PROCESSING

The following formats illustrate the standard text entries developed by phase 15 and phase 20 for the various types of operators. When the fields of the text entries differ from the standard definitions of the fields, the contents of the fields are explained. In addition, notes that explain the types of instructions generated by phase 25 are also included to the right of the text entry format, when appropriate. For an explanation of the individual operators see Table 29.

Table 31. Status Field Bits and Their Meanings

Operand/ Base Address	Bit	Meaning
	0	1 - subscript text item has been examined but not completely processed (internal to Register Optimization)
	1	1 - text item contains inert variable. Set by Register Optimization and used for communication with Text Optimization; text item is to be ignored.
Operand 2 base address status	2	0 - base address in storage 1 - base address in register
	3	0 - do not retain base address in register 1 - retain base address in register
Operand 3 base address status	4	0 - base address in storage 1 - base address in register
	5	0 - do not retain base address in register 1 - retain base address in register
Operand 2 status	6	0 - operand in storage 1 - operand in register
	7	0 - do not retain operand in register 1 - retain operand in register
Operand 3 status	8	0 - operand in storage 1 - operand in register
	9	0 - do not retain operand in register 1 - retain operand in register
Operand 1 base address status	10	0 - base address in storage 1 - base address in register
	11	0 - do not retain base address in register 1 - retain base address in register
Operand 1 status	12	0 - generate store into operand 1 1 - do not generate store into operand 1
	13	1 - if bits 6 & 7 are set to 1 and bit 12 is set to 0, generate register to register load in addition to store.
	14	1 - divide item actually MOD function (set and used by Register Optimization). If FC=44 or 15, load addresses precede.
	15	1 - .QXX temporary created for this item

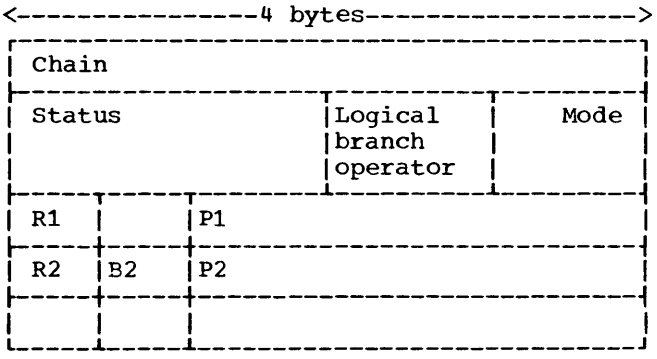
Branch Operator (B)



P1: The P1 field contains a pointer to the statement number/array table entry for the statement number to which a branch was made.

Note: Phase 25 decides whether an RR or an RX branch instruction should be generated.

Logical Branch Operators (BT, BF)

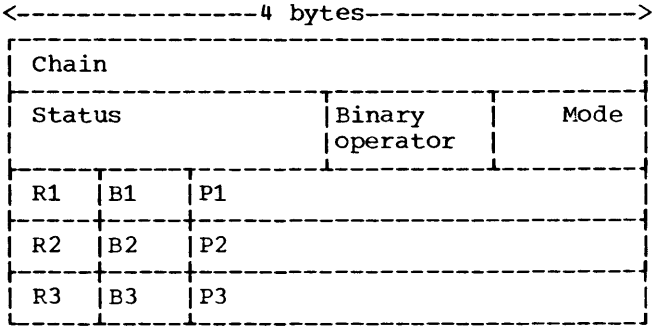


P1: The P1 field contains a pointer to the statement number/array table entry for the statement number to which a branch is being made.

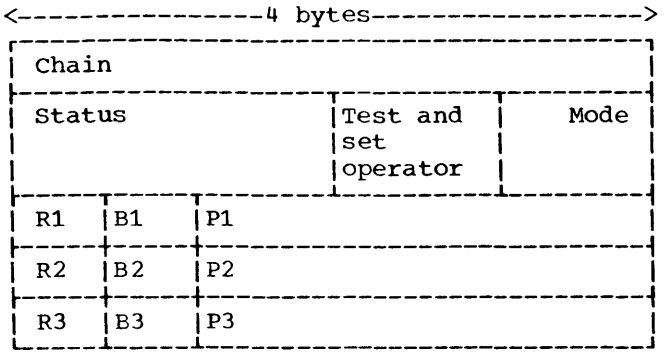
P2: The P2 field contains a pointer to the dictionary entry for the logical variable being tested.

Note: The test of the logical variable will be done with a BXH or BXLE for BT and BF, respectively.

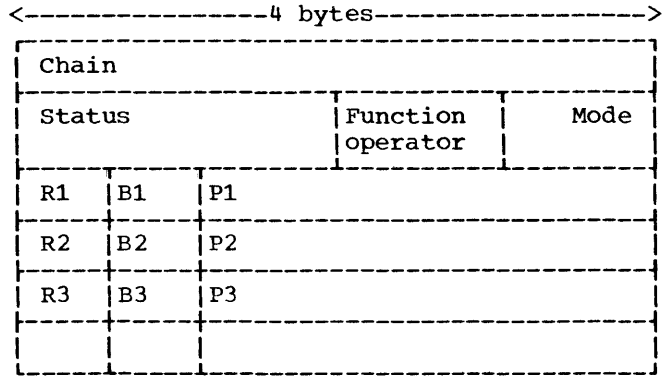
Binary Operators (+, -, *, /, OR, and AND)



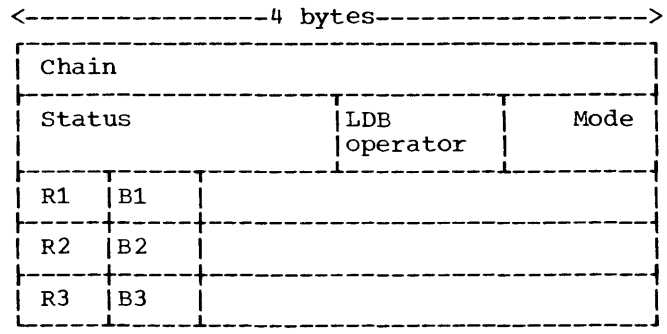
Test and Set Operators (GT, LT, GE, LE, EQ, and NE)



In-line Functions (MAX2, MIN2, DIM, IDIM, DMOD, MOD, AMOD, DSIGN, SIGN, ISIGN, LAND, LOR, LCOMPL, IDIM, BITON, BITOFF, AND, OR, COMPL, MOD24, SHFTR, and SHFTL)

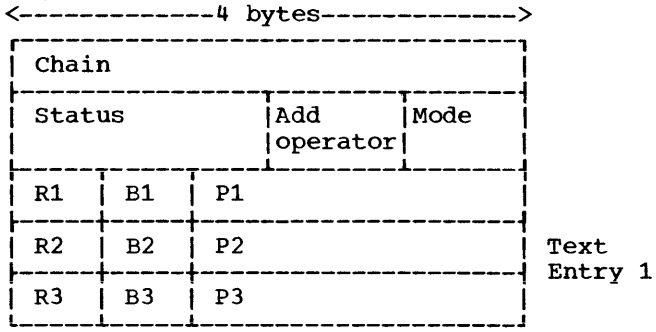


Testing a Byte Logical Variable (LDB)



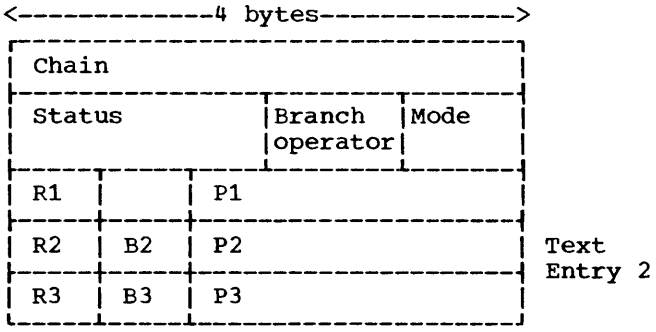
Note: The LDB operator is used to load a register with a byte logical variable.

Branch on Index Low or Equal, or Branch on Index High



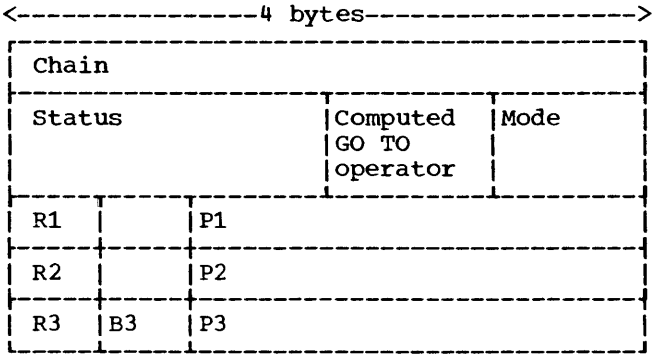
Note: A BXHLE instruction will be generated by phase 25 when an add operator is followed by a branch operator.

P1 and P2 of text entry 1 equals P2 of text entry 2.



P1: The P1 field of text entry 2 contains a pointer to the statement number/array table entry for the statement number to which a branch is being made.

Computed GO TO Operator

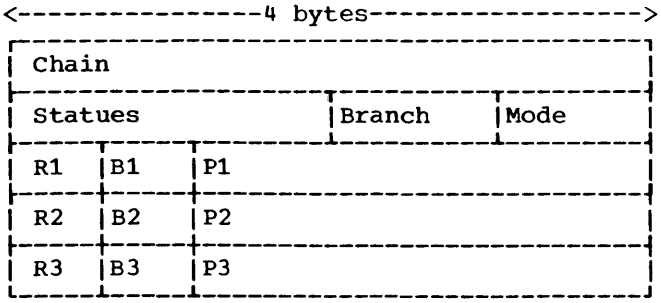


P1: P1 contains the number of items in the branch table that are associated with the computed GO TO operator.

P2: P2 contains a pointer to the information table entry for the branch table.

P3: P3 contains a pointer to the indexing value for the computed GO TO statement.

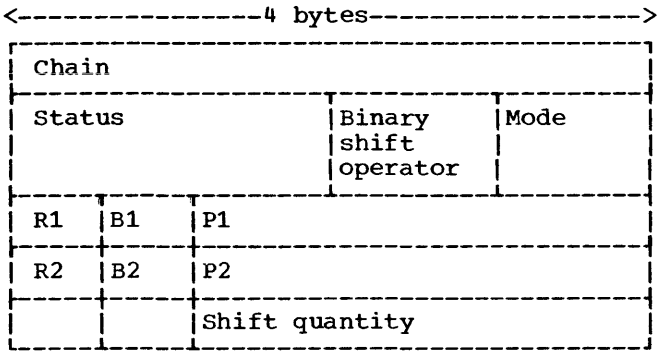
Branch Operators (BL, BLE, BE, BNE, BGE, BG, BLZ, BLEZ, BEZ, BNEZ, BGEZ, and BGZ)



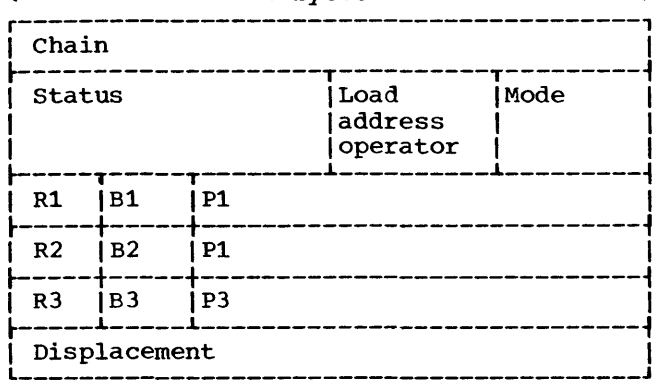
P1: The P1 field contains a pointer to the statement number/array table entry for the statement number to which a branch is being made.

Note: Operands 2 and 3 must be compared before the branch. For the BLZ, BLEZ, BEZ, BNEZ, BGEZ, and BGZ operators, operand 3 is zero and a test on zero is generated.

Binary Shift Operators (RS, LS)



Load Address Operator (LA)

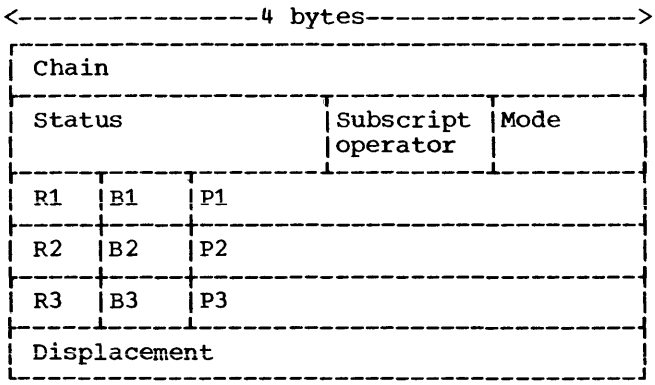


Note: The purpose of the load address operator is to store an address of an element of an array in a parameter list. If bit 7 of the status field is 1, the LA stores the last argument into the parameter list.

The P1 field points to a dictionary entry which points to the adcon table.

LA (14) is always followed by CALL (15) or a library function (44).

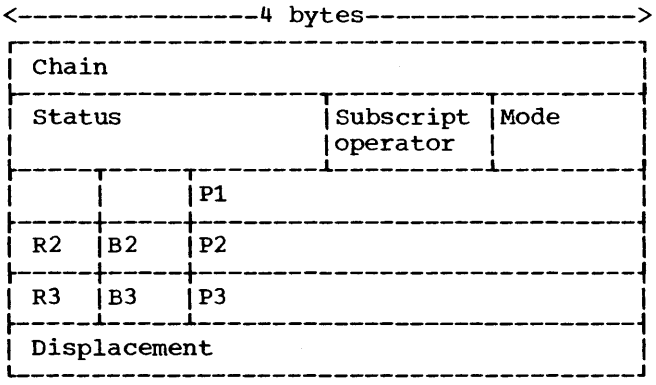
Subscript Text Entry -- Case 1



P2: The P2 field contains a pointer to the dictionary entry for the variable being indexed.

P3: The P3 field contains a pointer to the dictionary entry for the indexing value unless the indexing value is a constant; then P3 ≠ 0 and the displacement field contains a displacement.

Subscript Text Entry -- Case 2



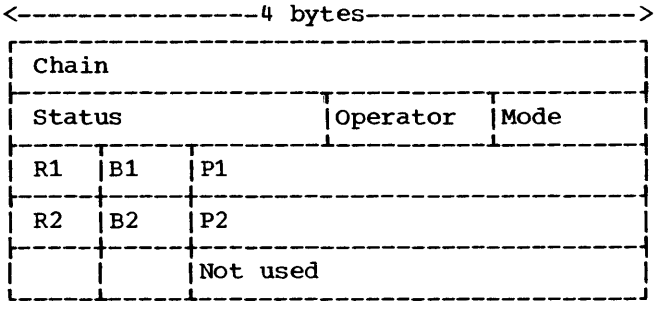
Note: For Case 2 subscript text entries, the subscript text entry is combined with the next text entry to form a single RX instruction. (Case 2 will be formed by phase 15 only when the second text entry has the store operator. Phase 20 will change Case 1 text entries to Case 2 text entries when appropriate.)

P1 is zero and either P2 or P3 of the next text entry will be zero.

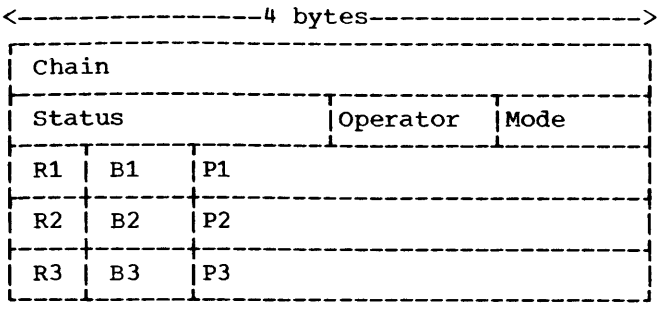
If the operator of the next text entry is a store, the subscript applies to P1. If the next operator is not a store, the subscript applies to operand = 0.

If the next operator is a 'LIST,' the subscript applies to P1 for READ or to P2 for WRITE.

In-line routines (DABS, ABS, IABS, IDINT, INT, HFIX, DFLOAT, FLOAT, DBLE)



EXT and LIBF Operators

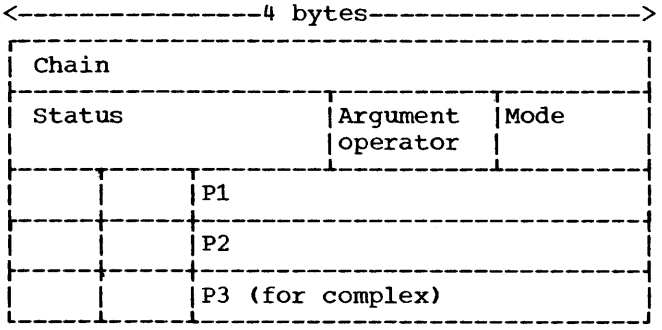


P1: P1 is zero for the EXT operator of a subroutine call.

P2: The P2 field contains either a pointer to the dictionary entry for an external function or a subroutine name, or a pointer to the IFUNTB entry for a library function.

P3: The P3 field contains either zero or a symbolic register number and a displacement that points to the object-time parameter list of the external function, library function, or subroutine.

Arguments for Functions and Calls

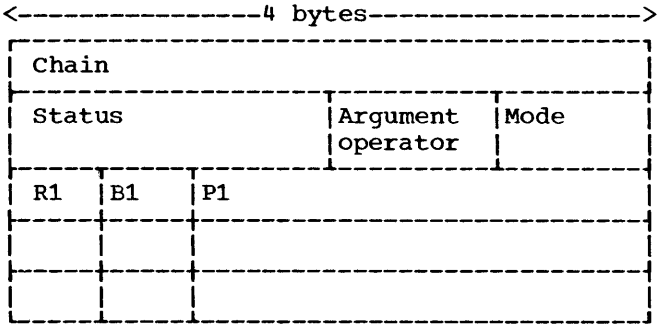


Note: No registers are needed for this type of text entry.

For calls and ABNORMAL functions, P1 = P2. For NORMAL functions and library functions, P1 = 0.

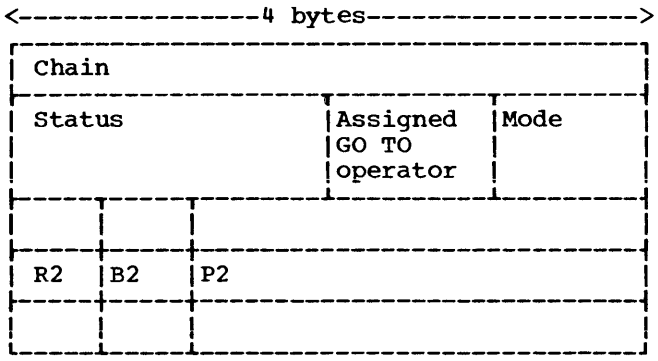
See the next text entry for the case of complex statements.

Special Argument Text Entry for Complex Statements



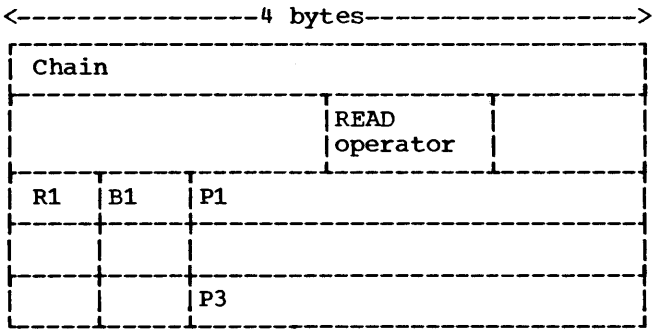
Note: For complex statements, the first text entry of the argument list contains the register information for the imaginary part of the complex result.

Assigned GO TO Operator (BA)



P2: The P2 field contains a pointer to the variable being used in the assigned GO TO statement.

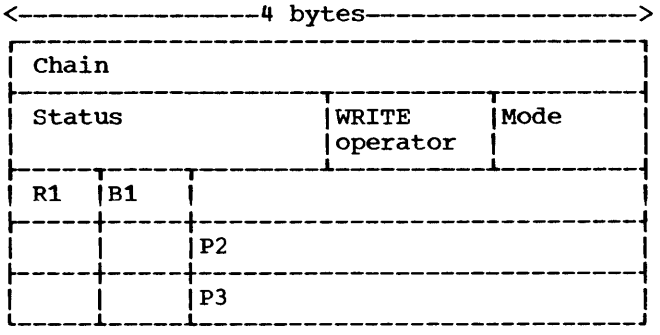
READ Operator for I/O List



P1: The P1 field contains a pointer to the I/O list for the READ statement. If this is an indexed READ, R1 is the register to be used.

Note: If the P3 field contains a nonzero, an entire array is being read. This causes a different instruction sequence to be generated.

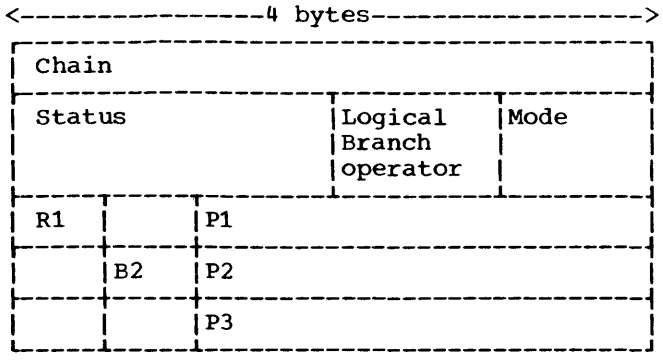
WRITE Operator for I/O List



P2: The P2 field contains a pointer to the I/O list for the WRITE statement. R1 and B1 are the index and base registers to be used for the WRITE.

Note: If the P3 field contains a nonzero, an entire array is being written. This causes a different instruction sequence to be generated.

Logical Branch Operators (BBT, BBF)

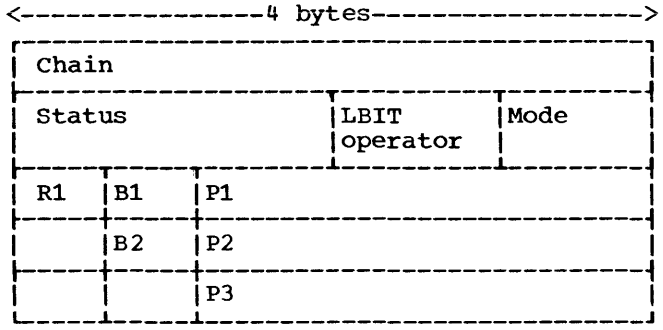


P1: The P1 field contains a pointer to the statement number/array table entry for the statement number to which a branch is being made.

P2: The P2 field contains a pointer to the dictionary entry for the logical variable being tested.

P3: The P3 field contains a pointer to the dictionary entry for the number of the bit being tested.

LBIT Operator



P2: The P2 field contains a pointer to the dictionary entry for the logical variable being tested.

P3: The P3 field contains a pointer to the dictionary entry for the number of the bit being tested.

The major arrays of the compiler are the bit-strip and skeleton arrays, which are used by phase 25 during code generation. The following illustrations detail the bit-strip and skeleton arrays associated with the operators of text entries that undergo code generation. The skeleton array for each operator is illustrated by a series of assembly language instructions, consisting of a basic operation code, which is modified to suit the mode of the operands, and by operands, which are in coded form. The operand codes and their meanings are, as follows:

Bn -- base register for operand n

BD -- base register used for loading an operand's base address

Rn -- operational register for operand n

X -- index register when necessary

To the right of the skeleton array for an operator is the bit-strip array for the operator. Each bit strip in the bit-strip array consists of a vertical string of 0's, 1's, and X's. A particular strip is selected according to the status information, which is shown above that strip. For example, if the combined status of operands 2 and 3 is 1010 (reading downward), the bit strip under that status is to be used during code generation. (The status of operand 2 is indicated in the first two vertical positions, reading downward; the status of operand 3 is indicated in the second two vertical positions, reading downward.[1]) The meanings of the various bit settings in each bit strip are, as follows:

0 -- The associated skeleton array instruction is not to be included as part of the machine code sequence. If a horizontal line containing all zeros appears after an instruction in a skeleton, the zero may be changed to a one to perform the desired function. This usually happens for base register loads and result stores.

1 -- The associated skeleton array instruction is to be included as part of the machine code sequence.

X -- The associated skeleton instruction may or may not be included as part of the machine code sequence, depending upon whether or not the associated base address is to be loaded, or whether or not a store into operand 1 is to be performed.

Note 1. KK is an indexing parameter used by Phase 25 which has a unique value for each skeleton.

Note 2. FC refers to the Phase 15/20 operators in Table 29.

IEKVPL: Used for All Subtract Operations

Index	Skeleton Instructions		Status
			0000000011111111
			0000111100001111
			0011001100110011
			0101010101010101
1	L	B2,D(0,BD)	XXXXXXXX00000000
2	LH	R2,D(0,B2)	0000111100000000
3	LH	R1,D(X,B2)	1100000000000000
4	L	B3,D(0,BD)	XX00XX00XX00XX00
5	LCR	R3,R3	0010001000000010
6	LR	R1,R2	0000110100001101
7	LH	R3,D(0,B3)	0100010001000100
8	LCR	R1,R3	0001000000000000
9	SH	R1,D(X,B3)	1000100010001000
10	SR	R1,R3	0100010101110101
11	AH	R3,D(X,B2)	0010000000000000
12	AH	R1,D(X,B2)	0001000000000000
13	AR	R3,R2	0000001000000010
14	L	B1,D(0,BD)	XXXXXXXXXXXXXXXX
15	STH	R1,D(0,B1)	XXXXXXXXXXXXXXXX

[1]In some cases, operand 3 does not exist and only the status of operand 2 is indicated.

IEKVTS: Used for the INT, IDINT, IFIX, and HFIX In-Line Routines

Index	Skeleton Instructions	INT, IFIX, HFIX Status	IDINT Status
		0011	0011
		0101	0101
1	SDR 0,0	1111	0000
2	L B2,D(0,BD)	XX00	XX00
3	LD R2,D(0,B2)	0100	0100
4	LD 0,D(0,B2)	1000	1000
5	LDR 0,R2	0111	0111
6	AW 0,60(0,12)	1111	1111
7	STD 0,64(0,13)	1111	1111
8	L R1,68(0,13)	1111	1111
9	BALR 15,0	1111	1111
10	BC 10,6(0,15)	1111	1111
11	LNR R1,R1	1111	1111
12	L B1,D(0,BD)	XXXX	XXXX
13	STH R1,D(0,B1)	XXXX	XXXX

IEKVAD: Used for the ABS (FC=67), IABS (FC=68), and DABS (FC=66) In-Line Routines (KK=25)

Index	Skeleton Instructions	Status
		0011
		0101
1	L B2,D(0,BD)	XX00
2	LH R2,D(0,B2)	1100
3	LPR R1,R2	1111
4	L B1,D(0,BD)	XXXX
5	STH R1,D(0,B1)	XXXX

IEKVFP: Used for the MOD24 In-Line Routine

Index	Skeleton Instructions	Status
		0011
		0101
1	L B2,D(0,BD)	XX00
2	L R2,D(X,B2)	1100
3	LA R1,0(0,R2)	1111
4	L B1,D(0,BD)	XXXX
5	ST R1,D(0,B1)	XXXX

IEKVTS: Used for the MAX2 and MIN2 In-Line Routines

Index	Skeleton Instructions	Status
		0000000011111111
		0000111100001111
		0011001100110011
		0101010101010101
1	L B2,D(0,BD)	XXXXXXXX00000000
2	LH R2,D(0,B2)	0000111100000000
3	LH R1,D(0,B2)	1100000000000000
4	CR R1,R2	0000001000000010
5	CH R3,D(0,B2)	0001000000000000
6	CH R1,D(0,B2)	0010000000000000
7	L B3,D(0,BD)	XX00XX00XX00XX00
8	LH R3,D(0,B3)	0100010001000100
9	CR R2,R3	0100010101110101
10	CH R2,D(0,B3)	0000100000001000
11	CH R1,D(0,B3)	1000000010000000
12	LR R1,R2	0000110100001101
13	LR R1,R3	0001000000000000
14	BALR 15,0	1111111111111111
15	BC N,6(0,15)[1]	1111111111111111
16	LR R1,R2	0000001000000010
17	LR R1,R3	0100010101110101
18	LH R1,D(0,B2)	0011000000000000
19	LH R1,D(0,B3)	1000100010001000
20	L B1,D(0,BD)	XXXXXXXXXXXXXXXX
21	STH R1,D(0,B1)	XXXXXXXXXXXXXXXX

[1]For MAX2,N=2; for MIN2,N=4.

IEKVFP: Used for the SHFTR and SHFTL In-Line Routines

Index	Skeleton Instructions	Status
		0000000011111111
		0000111100001111
		0011001100110011
		0101010101010101
1	L B2,D(0,BD)	XXXXXXXX00000000
2	L R2,D2(X,B2)	1111111100000000
3	LR R1,R2	0000111100001111
4	L B3,D(0,BD)	XX00XX00XX00XX00
5	LH R3,D3(X,B3)	1100110011001100
6	SRL R1,0(0,R3)	1111111111111111
7	L B1,D(0,BD)	XXXXXXXXXXXXXXXX
8	ST R1,D(0,B1)	XXXXXXXXXXXXXXXX

IEKVAD: Used for the DBLE In-Line Routines

Index	Skeleton Instructions		Status
			0011
			0101
1	L	B2,D(0,BD)	XX00
2	SDR	R1,R1	1111
3	LER	0,R2	0010
4	LE	R1,D(0,B2)	1100
5	LER	R2,R1	0100
6	LDR	R1,0	0010
7	LER	R1,R2	0001
8	L	B1,D(0,BD)	XXXX
9	STD	R1,D(0,B1)	XXXX

IEKVAD: Used for DMOD (FC=60) and AMOD (FC=62) In-Line Routines (KK=22)

Index	Skeleton Instructions		Status
			0000000011111111
			0000111100001111
			0011001100110011
			0101010101010101
1	L	B2,D(0,BD)	XXXXXXXX00000000
2	LD	R2,D(0,B2)	0000111100000000
3	LD	R1,D(0,B2)	1111000000000000
4	L	B3,D(0,BD)	XX00XX00XX00XX00
5	LD	R3,D(0,B3)	0100010001000100
6	LDR	R1,R2	0000111111111111
7	DDR	R1,R3	0111011101110111
8	DD	R1,D(0,B3)	1000100010001000
9	AD	R1,n(0,13)*	1111111111111111
10	MDR	R1,R3	0111011101110111
11	MD	R1,D(0,B3)	1000100010001000
12	LCDR	R1,R1	1111111111111111
13	AD	R1,D(0,B2)	1111111100000000
14	ADR	R1,R2	0000000011111111
15	L	B1,D(0,BD)	XXXXXXXXXXXXXXXX
16	STD	R1,D(0,B1)	XXXXXXXXXXXXXXXX

*Note that n is the displacement assigned by the compiler to the constant 4E00000000000000. Note also that this instruction is generated twice with the operation code changed to AW for the first of the two generations.

IEKVTS: Used for SIGN, ISIGN, and DSIGN In-Line Routines

Index	Skeleton Instructions		Status
			0000000011111111
			0000111100001111
			0011001100110011
			0101010101010101
1	L	B2,D(0,BD)	XXXXXXXX00000000
2	LH	R2,D(0,B2)	0000111100000000
3	LTR	R3,R3	0010001000100010
4	LH	R1,D(0,B2)	1111000000000000
5	L	B3,D(0,BD)	XX00XX00XX00XX00
6	LH	R3,D(0,B3)	0100010001000100
7	LR	R1,R2	0000001000000010
8	LPR	R1,R2	0000110100001101
9	LPR	R1,R1	1101000011010000
10	LTR	R3,R3	0101010101010101
11	TM	128,D(0,B3)	1000100010001000
12	BALR	15,0	1111111111111111
13	BC	14,6(0,15)	1000100010001000
14	BC	10,6(0,15)	0111011101110111
15	LR	R1,R1	1111111111111111
16	BC	15,12(0,15)	0010001000100010
17	LPR	R1,R1	0010001000100010
18	L	B1,D(0,BD)	XXXXXXXXXXXXXXXX
19	STH	R1,D(0,B1)	XXXXXXXXXXXXXXXX

IEKVTS: Used for DIM and IDIM In-Line Routines

Index	Skeleton Instructions		Status
			0000000011111111
			0000111100001111
			0011001100110011
			0101010101010101
1	L	B2,D(0,BD)	XXXXXXXX00000000
2	LH	R2,D(0,B2)	0000111100000000
3	LH	R1,D(0,B2)	1101000000000000
4	LCR	R1,R3	0010001000000010
5	AH	R1,D(0,B2)	0010000000000000
6	L	B3,D(0,BD)	XX00XX00XX00XX00
7	LH	R3,D(0,B3)	0100010001000100
8	LR	R1,R2	0000110100001101
9	SH	R1,D(0,B3)	1000100010001000
10	AR	R1,R2	0000001000000010
11	SR	R1,R3	0101010101110101
12	BALR	15,0	1111111111111111
13	BC	10,6(0,15)	1111111111111111
14	SR	R1,R1	1111111111111111
15	L	B1,D(0,BD)	XXXXXXXXXXXXXXXX
16	STH	R1,D(0,B1)	XXXXXXXXXXXXXXXX

IEKVUN: Used for NOT Operations

Index	Skeleton Instructions		Status
			0011
			0101
1	L	B2,D(0,BD)	XX00
2	LA	R1,1(0,0)	1101
3	BCTR	R1,0	0010
4	LCR	R1,R1	0010
5	X	R1,D(X,B2)	1000
6	L	R2,D2(0,B2)	0100
7	XR	R1,R2	0101
8	L	B1,D(0,BD)	XXXX
9	ST	R1,D(0,B1)	XXXX

IEKVUN: Used for All Load Address Operations

Index	Skeleton Instructions		Status
			0000000011111111
			0000111100001111
			0011001100110011
			0101010101010101
1	L	B3,D(0,BD)	0000000000000000
2	LH	R3,D(0,B3)	1100110011001100
3	L	B2,D(0,BD)	0000000000000000
4	LA	R1,D(R3,B2)	1111111111111111
5	L	B1,D(0,BD)	0000000000000000
6	ST	R1,D(0,B1)	1111111111111111
7	LA	0,128(0,0)	0000000000000000
8	MVI	128,D(0,B1)	0000111100000000

IEKVUN: Used for All Load Byte Operations

Index	Skeleton Instructions		Status
			0000000011111111
			0000111100001111
			0011001100110011
			0101010101010101
1	L	B3,D(0,BD)	0000000000000000
2	SR	R3,R3	1111111100000000
3	IC	R3,D(X,B3)	1111111111111111
4	L	B1,D(0,BD)	0000000000000000
5	ST	R3,D(0,B1)	0000000000000000

IEKVAD: Used for COMPL and LCOMPL In-Line Routines

Index	Skeleton Instructions		Status
			0011
			0101
			0000
			0000
1	L	B2,D(0,BD)	XX00
2	L	R2,D(0,B2)	0100
3	LA	R1,1(0,0)	1101
4	LCR	R1,R1	1111
5	X	R1,D2(X,B2)	1000
6	XR	R1,R2	0101
7	BCTR	R1,0	0010
8	L	B1,D(0,BD)	XXXX
9	ST	R1,D(0,B1)	XXXX

IEKVBL: Used for All Branch True and Branch False Operations

Index	Skeleton Instructions		Status
			0000000011111111
			0000111100001111
			0011001100110011
			0101010101010101
1	L	B2,D(0,BD)	0000000000000000
2	L	R2,D(0,B2)	1111111100000000
3	SR	R3,R3	1100110011001100
4	L	B1,D(0,BD)	1111111111111111
5	BXH	R2,0(R3,B1)	1111111111111111*
6	BXLE	R2,0(R3,B1)	1111111111111111*

*One of these two instructions will be added to the bit strip by subroutine MAINGN-IEKTA depending on the operation.

IEKVPL: Used for all Half-Word Integer Division Operations and for the MOD In-Line Routine

Index	Skeleton Instructions		Status
			0000000011111111
			0000111100001111
			0011001100110011
			0101010101010101
1	L	B2,D(0,BD)	0000000000000000
2	LH	R2,D(0,B2)	0000111100000000
3	LH	R1,D(0,B2)	1111000000000000
4	L	B3,D(0,BD)	0000000000000000
5	LH	R3,D(X,B3)	1100110011001100
6	LR	R1,R2	0000111100001111
7	SRDA	R1,32(0,0)	1111111111111111
8	DR	R1,R3	1111111111111111
9	D	R1,D(X,B3)	0000000000000000
10	L	B1,D(0,BD)	0000000000000000
11	STH	R1+1,D(0,B1)	0000000000000000
12	STH	R1,D(0,B1)*	0000000000000000

*For MOD in-line routine only.

IEKVSU: Used for Case 1 and Case 2 Subscript Operations

Index	Skeleton Instructions		Status
Case 1			
			0000000011111111
			0000111100001111
			0011001100110011
			0101010101010101
1	L	B3,D(0,BD)	0000000000000000
2	LH	R3,D(0,B3)	1100110000000000
3	L	B2,D(0,BD)	0000000000000000
4	LH	R2,D(R3,B2)	1111111100000000
5	L	B1,D(0,BD)	0000000000000000
6	STH	R2,D(0,B1)	0000000000000000
Case 2			
			0000000011111111
			0000111100001111
			0011001100110011
			0101010101010101
1	L	B3,D(0,BD)	0000000000000000
2	LH	R3,D(0,B3)	1100110011001100
3	L	B2,D(0,BD)	0000000000000000
4	LH	R2,D(R3,B2)	0000000000000000
5	L	B1,D(0,BD)	0000000000000000
6	STH	R2,D(0,B1)	0000000000000000

IEKVUN: Used for All Unary Minus Operations

Index	Skeleton Instructions		Status
			0000000011111111
			0000111100001111
			0011001100110011
			0101010101010101
1	L	B2,D(0,BD)	0000000000000000
2	LH	R2,D2(X,B2)	1111111100000000
3	LCR	R1,R2	1111111111111111
4	L	B1,D(0,BD)	0000000000000000
5	STH	R1,D1(X,B1)	0000000000000000

IEKVBL: Used for All Assigned GO TO Operations

Index	Skeleton Instructions		Status
			0000000011111111
			0000111100001111
			0011001100110011
			0101010101010101
1	L	B2,D(0,BD)	0000000000000000
2	L	R2,D(0,B2)	1111111100000000
3	BCR	15,R2	1111111111111111

IEKVBL: Used for All Computed GO TO Operations

Index	Skeleton Instructions		Status
			0000000011111111
			0000111100001111
			0011001100110011
			0101010101010101
1	L	B3,D(0,BD)	0000000000000000
2	L	R3,D3(0,B3)	1100110011001100
3	LR	R1,R3	0101010101010101
4	LA	R2,P1(0,0)	1111111111111111
5	CLR	R1,R2	1111111111111111
6	BALR	R2,0	1111111111111111
7	SLL	R1,2(0,0)	1111111111111111
8	BC	2,14(0,R2)	1111111111111111
9	L	R2,D(R1,B)	1111111111111111
10	BCR	15,R2	1111111111111111

IEKVSU: Used for All Store Operations

Index	Skeleton Instructions		Status
			0000000011111111
			0000111100001111
			0011001100110011
			0101010101010101
1	L	B2,D(0,BD)	0000000000000000
2	LH	R2,D(0,B2)	1111111100001000
3	L	B1,D(0,BD)	0000000000000000
4	STH	R2,D(X,B1)	0000000000000000

IEKVAD: Used for the AND and OR In-Line Routines

Index	Skeleton Instructions		Status
			0000000011111111
			0000111100001111
			0011001100110011
			0101010101010101
1	L	B2,D(0,BD)	0000000000000000
2	L	R1,D(X,B2)	1111111100000000
3	L	B3,D(0,BD)	0000000000000000
4	N	R1,D(X,B3)	1111111111111111
5	L	B1,D(0,BD)	0000000000000000
6	ST	R1,D(0,B1)	0000000000000000

IEKVSU: Used for All Right- and Left-Shift Operations

Index	Skeleton Instructions		Status
			0000000011111111
			0000111100001111
			0011001100110011
			0101010101010101
1	L	B2,D(0,BD)	0000000000000000
2	LH	R2,D(0,B2)	1111111100000000
3	LR	R1,R2	0000111100001111
4	SRA	R1,P3(0,0)	1111111111111111
5	HDR	R1,R2	0000000000000000
6	L	B1,D(0,BD)	0000000000000000
7	STH	R1,D(0,B1)	0000000000000000

IEKVTS: Used for the FLOAT and DFLOAT In-Line Routines

Index	Skeleton Instructions		Status
			0011
			0101
1	L	B2,D(0,BD)	XX00
2	LH	R2,D(0,B2)	1100
3	LD	R1,60(0,12)	1111
4	STD	R1,72(0,13)	1111
5	LTR	R2,R2	1111
6	BALR	15,0	1111
7	BC	4,16(0,15)	1111
8	ST	R2,76(0,13)	1111
9	AD	R1,72(0,13)	1111
10	BC	15,26(0,15)	1111
11	LPR	0,R2	1111
12	ST	0,76(0,13)	1111
13	SD	R1,72(0,13)	1111
14	L	B1,D(0,BD)	XXXX
15	STD	R1,D(0,B1)	XXXX

IEKVPL: Used for All Fixed Point Multiplication Operations

Index	Skeleton Instructions		Status
			0000000011111111
			0000111100001111
			0011001100110011
			0101010101010101
1	L	B2,D(0,BD)	0000000000000000
2	LH	R2,D(0,B2)	0000111100000000
3	LH	R1,D(X,B2)	1100000000000000
4	L	B3,D(0,BD)	0000000000000000
5	LH	R3,D(0,B3)	0100010001000100
6	LR	R1,R2	0000110100001101
7	LR	R1,R3	0001000000000000
8	MR	R1-1,R3	0100010101110101
9	MR	R1-1,R2	0000001000000010
10	MH	R1,D(X,B3)	1000100010001000
11	MH	R1,D(X,B2)	0011000000000000
12	L	B1,D(0,BD)	0000000000000000
13	STH	R1,D(0,B1)	0000000000000000

IEKVPL: Used for all Full-Word Integer Division Operations and for the MOD In-Line Routine

Index	Skeleton Instructions		Status
			0000000011111111
			0000111100001111
			0011001100110011
			0101010101010101
1	L	B2,D(0,BD)	0000000000000000
2	LH	R2,D(0,B2)	0000111100000000
3	LH	R1,D(0,B2)	1111000000000000
4	L	B3,D(0,BD)	0000000000000000
5	LH	R3,D(X,B3)	0100010001000100
6	LR	R1,R2	0000111100001111
7	SRDA	R1,32(0,0)	1111111111111111
8	DR	R1,R3	0111011101110111
9	D	R1,D(X,B3)	1000100010001000
10	L	B1,D(0,BD)	0000000000000000
11	STH	R1+1,D(0,B1)	0000000000000000
12	STH	R1,D(0,B1)*	0000000000000000

* For MOD in-line routine only.

IEKVUN: Used for All Logical Operations

Index	Skeleton Instructions		Status
			0000000011111111
			0000111100001111
			0011001100110011
			0101010101010101
1	L	B2,D(0,BD)	0000000000000000
2	L	R2,D(0,B2)	0000111100000000
3	L	R1,D2(0,B2)	1101000000000000
4	L	B3,D(0,BD)	0000000000000000
5	L	R3,D(0,B3)	0100010001000100
6	L	R1,D3(X,B3)	0000100000001000
7	LR	R1,R2	0000010100000101
8	NR	R1,R2	0000101000001010
9	NR	R1,R3	0101010101110101
10	N	R1,D2(0,B2)	0010000000000000
11	N	R1,D3(X,B3)	1000000010000000
12	L	B1,D(0,BD)	0000000000000000
13	ST	R1,D1(0,B1)	0000000000000000

IEKVTS: Used to Compare Operands Across a Relational Operator and Set the Result to True or False

Index	Skeleton Instructions		Status
			0000000011111111
			0000111100001111
			0011001100110011
			0101010101010101
1	L	B2,D(0,BD)	0000000000000000
2	LH	R2,D(X,B2)	1111111100000000
3	L	B3,D(0,BD)	0000000000000000
4	LH	R3,D(0,B3)	0100010001000100
5	CH	R2,D(X,B3)	1000100010001000
6	CR	R2,R3	0111011101110111
7	LA	R1,1(0,0)	1111111111111111
8	BALR	15,0	1111111111111111
9	BC	M,6(0,15)	1111111111111111
10	SR	R1,R1	1111111111111111
11	L	B1,D(0,BD)	0000000000000000
12	ST	R1,D(0,B1)	0000000000000000

IEKVPL: Used for All Addition Operations and for Real Multiplication and Real Division Operations

Index	Skeleton Instructions		Status
			0000000011111111
			0000111100001111
			0011001100110011
			0101010101010101
1	L	B2,D(0,BD)	XXXXXXXX00000000
2	LH	R2,D(0,B2)	0000111100000000
3	LH	R1,D(X,B2)	1101000000000000
4	L	B3,D(0,BD)	XX00XX00XX00XX00
5	LH	R3,D(0,B3)	0100010001000100
6	LH	R1,D(X,B3)	0000000000000000
7	LR	R1,R2	0000110100001101
8	AR	R1,R2	0000001000000010
9	AR	R1,R3	0101010101110101
10	AH	R1,D(X,B2)	0010000000000000
11	AH	R1,D(X,B3)	1000100010001000
12	L	B1,D(0,BD)	XXXXXXXXXXXXXXXX
13	STH	R1,D(0,B1)	XXXXXXXXXXXXXXXX

Note: For real multiplication and division operations, the basic operation codes will be replaced by the required codes.

IEKVBL: Used for Text Entries Whose Operator is a Relational Operator Operating on Two Nonzero Operands

Index	Skeleton Instructions		Status
			0000000011111111
			0000111100001111
			0011001100110011
			0101010101010101
1	L	B2,D(0,BD)	0000000000000000
2	LH	R2,D(0,B2)	1111111100000000
3	L	B3,D(0,BD)	0000000000000000
4	LH	R3,D(X,B3)	0100010001000100
5	CH	R2,D(X,B3)	1000100010001000
6	CR	R2,R3	0111011101110111
7	LTR	R2,R2	0000000000000000
8*	L	R1,P1	1111111111111111
9	BCR	M,R1	1111111111111111

*IEKVBL will generate instruction 8 only if P1 points to a B-block.

IEKVBL: Used for Text Entries Whose Operator is a Relational Operator Operating on One Operand and Zero

Index	Skeleton Instructions		Status
			0000000011111111
			0000111100001111
			0011001100110011
			0101010101010101
1	L	B2,D(0,BD)	0000000000000000
2	LH	R2,D(0,B2)	1111111100000000
3	L	B3,D(0,BD)	0000000000000000
4	LH	R3,D(X,B3)	0000000000000000
5	CH	R2,D(X,B3)	0000000000000000
6	CR	R2,R3	0000000000000000
7	LTR	R2,R2	1111111111111111
8*	L	R1,P1	1111111111111111
9	BCR	M,R1	1111111111111111

*IEKVBL will generate instruction 8 only if P1 points to a B-block.

IEKVFP: Used for the LBIT, BBT, and BBF In-Line Routines

Index	Skeleton Instructions		BBT,BBF		LBIT	
			Simple Variable	Subscripted Variable	Simple Variable	Subscripted Variable
1	L	B2,D(0,BD)	X	X	X	X
2	LA	15,D+N/8(X,B2)	0	1	0	1
3	TM	M,D+N/8(B2)	1	0	1	0
4	TM	M,0(15)	0	1	0	1
5	TM	M,D+N/8(R2)	0	0	0	0
6	L	15,P1	1	1	0	0
7	BCR	MM,15	1	1	0	0
8	BALR	15,0	0	0	1	1
9	LA	R1,1(0,0)	0	0	1	1
10	BC	1,10(0,15)	0	0	1	1
11	SR	R1,R1	0	0	1	1
12	L	B1,D(0,BD)	0	0	X	X
13	ST	R1,D(0,B1)	0	0	X	X

N = The bit to be loaded or tested.

M = MSKTBL(MOD(N,8)+1). MSKTBL is an array of masks used by IEKVFP.

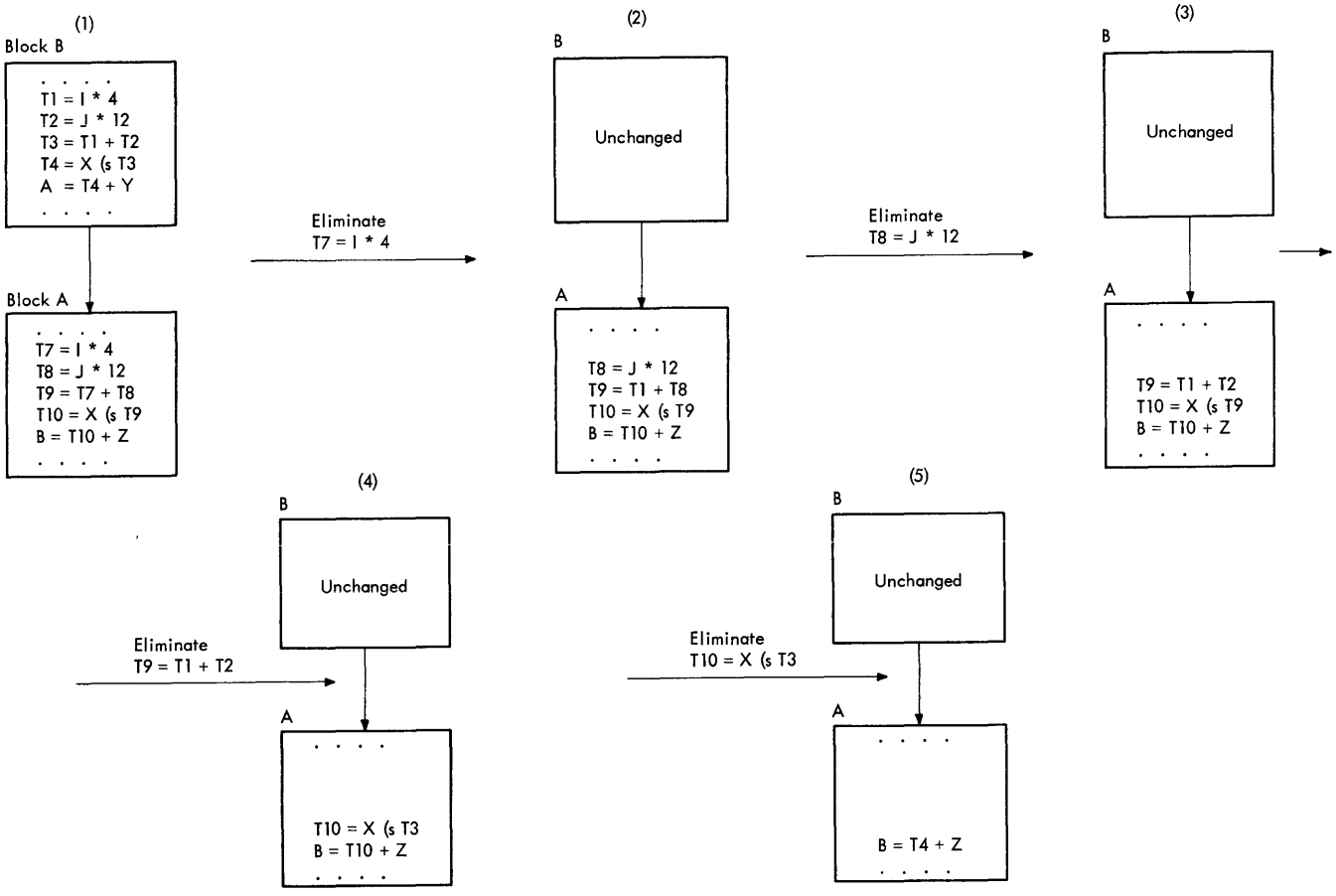
MM = 1 FOR BBT.

MM = 8 FOR BBF.

This appendix contains examples that illustrate the effects of text optimization on sample text entry sequences. An example is presented for each of the four sections of text optimization.

Example 1: Common Expression Elimination

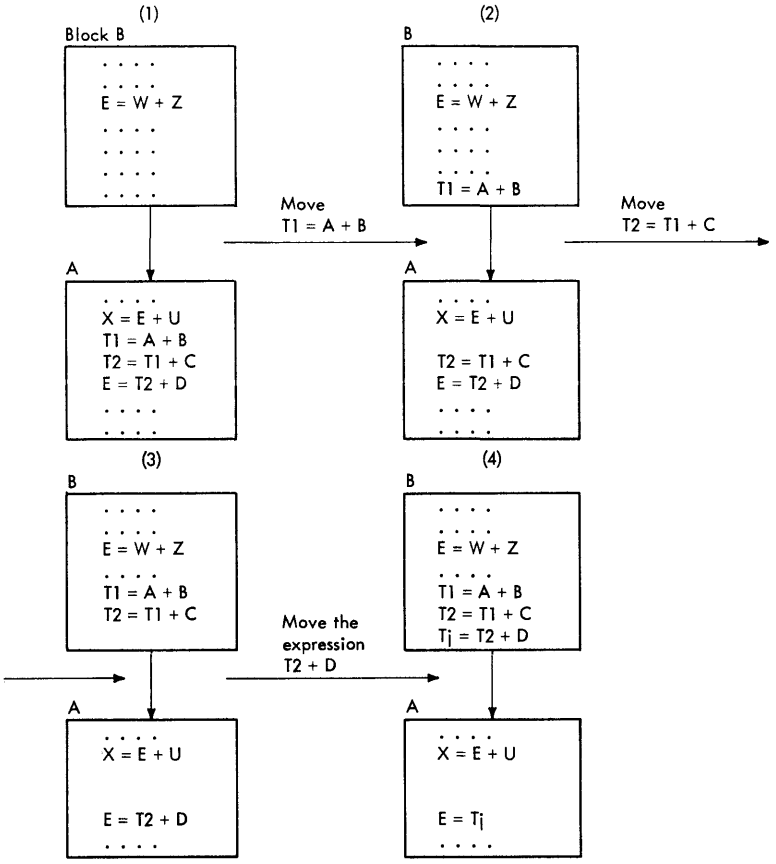
This example illustrates the concept of common expression elimination. The text entries in block A are to undergo common expression elimination. Block B is a back dominator of block A. Block B contains text entries that are common to those in block A.



NOTE: The items Ti are temporaries and (s represents a subscript operator

Example 2: Backward Movement

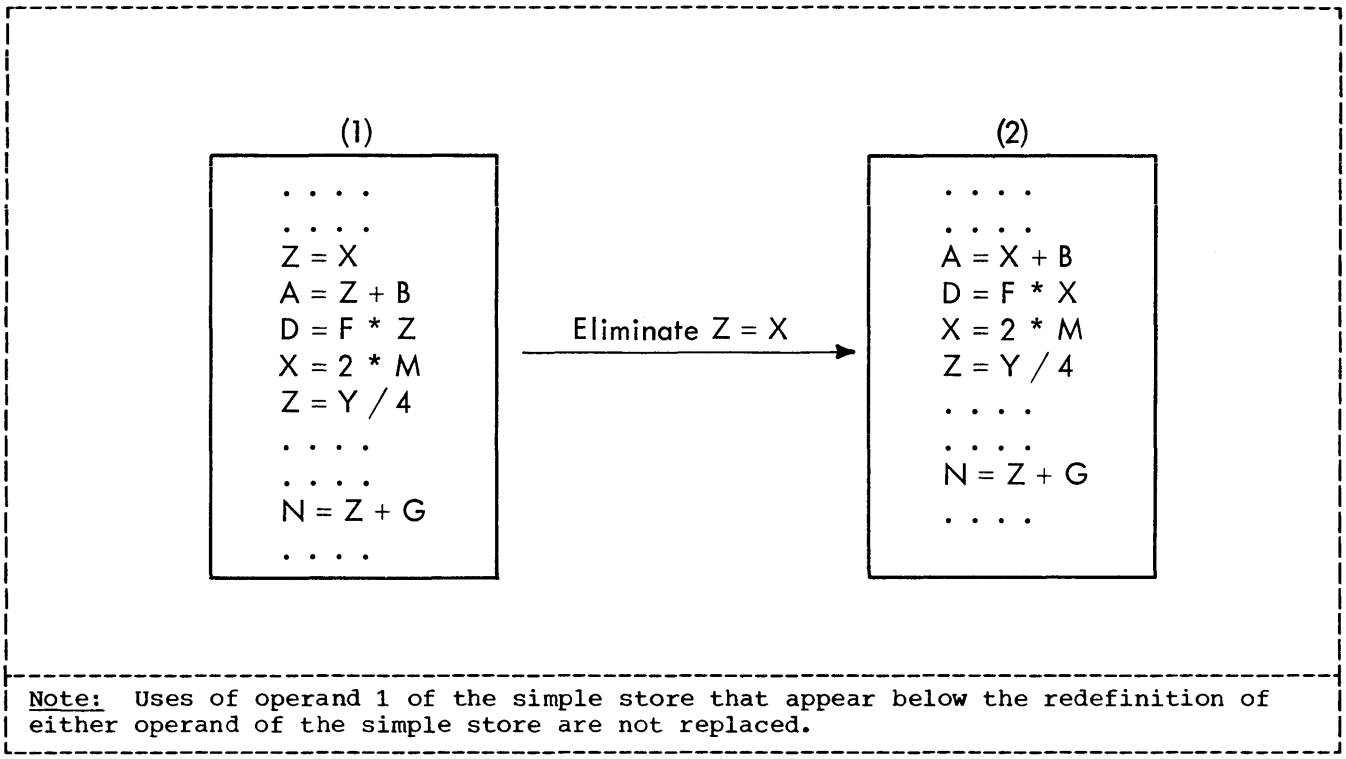
This example illustrates both methods of backward movement. The text entries in block A are to undergo backward movement. Block B is the back target of the loop containing block A.



NOTE: The text entry X = E + U cannot be moved, because its operand 2 is defined elsewhere in the loop. The text entry E = T2 + D cannot be moved, because operand 1 (E) is busy-on-exit from the back target; however, the expression T2 + D can be moved.

Example 3: Simple-Store Elimination

The following example illustrates the concept of simple-store elimination, an integral part of the processing of backward movement.



Note: Uses of operand 1 of the simple store that appear below the redefinition of either operand of the simple store are not replaced.

Example 4: Strength Reduction

This example illustrates both methods of strength reduction. In the example, strength reduction is applied to a DO loop. The evolution of the text entries that represent the DO loop and the functions of these text entries are also shown. The formats of the text entries in all cases are not exact. They are presented in this manner to facilitate understanding.

Consider the DO loop:

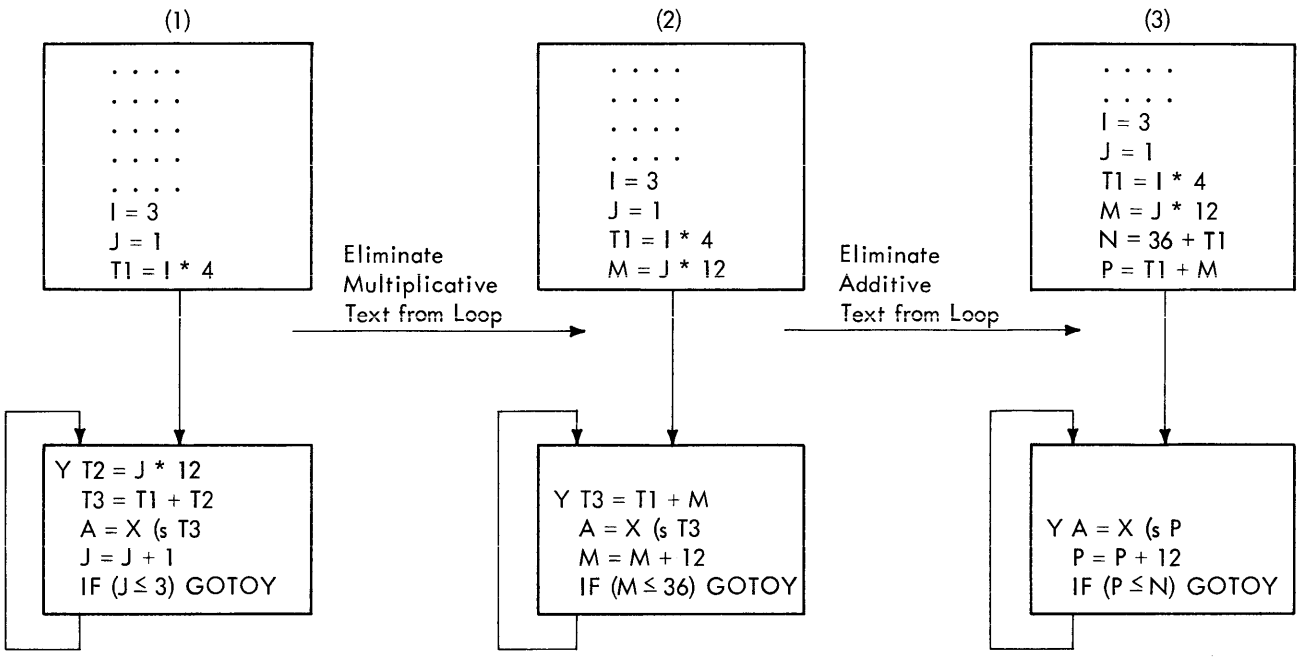
```
        I=3
        DO 10 J=1,3
        A=X(I,J)
   10   CONTINUE
```

As a result of the processing of phases 10 and 15, and backward movement, the DO loop has been converted to the following text representation.

	Text Entry	Function	Evolution
Back Target	I = 3	Initializes I	Stated in source module, converted to phase 10 text and then to phase 15 text. It resides in the back target of the loop because of text blocking.
	J = 1	Initializes J	Generated phase 10 text entry, converted to phase 15 text entry. It resides in the back target of the loop because of text blocking.
	T1 = I * 4	Multiplies first subscript parameter by its dimension factor	Generated by phase 15 when it encounters the subscript parameter I during its processing of phase 10 text. It resides in the back target of the loop as a result of the processing of backward movement.
Loop	Y T2 = J * 12	Multiplies second subscript parameter by its dimension factor.	Generated by phase 15 when it encounters the subscript parameter J during its processing of phase 10 text.
	T3 = T1 + T2	Computes index value for the subscripted variable X.	Generated by phase 15 after the last subscript parameter in the phase 10 text representation of the subscripted variable has been processed.
	A = X (s T3	Stores X(I,J) into A	The phase 10 text entry forced and converted to phase 15 text after the index value for the subscripted variable has been established.
	J = J + 1	Increments DO index.	Generated by phase 10 and converted to phase 15 text representation.
	IF(J≤3)GOTO Y	Tests DO index against its maximum and controls branching.	Generated by phase 10 and converted to phase 15 text representation.

Note: The statement number Y is generated by phase 10. Also, it is assumed that the array X is of the format X(3,3) and that its elements are real (length 4).

The following illustration shows the application of strength reduction to the loop.



APPENDIX E: ADDRESS COMPUTATION FOR ARRAY ELEMENTS

Data references in the form of subscripted variable expressions in FORTRAN are converted into object code that includes address arithmetic and indexed references to main storage addresses. Since the conversion involves all phases of the compiler, a summary of the method is given here.

Consider an array A of n dimensions whose element length is L, and whose dimensions are D1, D2, D3, ...,Dn. If such an array is assigned main storage starting at the address P11, then the element A(J1, J2, J3,...,Jn) is located at:

```
P = P11 + (J1-1)*L + (J2-1)*D1*L +
    (J3-1)*D1*D2*L + ... + (Jn-1)*D1*D2*D3*
    ...*D(n-1)*L
```

This may be expressed as:

```
P = P00 + J1*L + J2*(D1*L) + J3*(D1*D2*L)
    + ... + Jn*(D1*D2*D3* ... *D(n-1)*L)
```

where:

```
P00 = P11 - (L+D1*L + D1*D2*L + ... +
      D1*D2* ... *D(n-1)*L)
```

For fixed dimensioned arrays, the quantities D1*L, D1*D2*L, D1*D2*D3*L, ... , which are referred to as dimension factors, are computed at compile time. The sum of these quantities, which is referred to as the span of the array, is also computed at compile time. (Phase 15 assigns to an array a relative address equal to its actual relative address minus the span of the array.)

In the object code, P is finally formed as the sum of a base register, an index register, and a displacement. The phase 15 segment CORAL associates an address constant with each fixed dimensioned array such that Pa≤P00≤Pa+4095, where Pa is the address inserted into the address constant at program fetch time. The effective address is then formed using a base register containing the address constant, a displacement equal to P00 - Pa, and an index register, which contains the result of a computation of the form:

```
L     2,J1
SLL   2,log2L
L     1,J2
M     0,L*D1
AR    2,1
L     1,J3
M     0,D1*D2*L
AR    2,1
.
.
.
L     1,Jn
M     0,D1*D2*...*D(n-1)
AR    2,1
```

Absorption of Constants in Subscript Expressions

Subscript expressions may include constant parts whose contribution to the final effective address is computed at compile time. For example,

```
B(I-2,J+4,3*5-(L+7)-6)
```

would usually be treated in such a way that the effect of the 2, the 4, and the 6 would be absorbed into the displacement at compile time.

Consider an example of the form

```
A(J1+K1,J2+K2, ... ,Jn+Kn),
```

where:

A is a fixed dimensioned array
K1, K2, ..., Kn are integer constants

Phase 15 will insert the quantity

```
K1*L + K2*(D1*L) + K3*(D1*D2*L) +
   ... + Kn(D1*D2* ... *D(n-1)*L)
```

into the displacement (DP) field of the corresponding subscript or load address text entry. The constants will not otherwise be included in the subscript expression. When phase 25 generates machine code, the contents of the DP field are added to the displacement. To ensure that the resultant expression lies within the range of 0 to 4095, phase 20 performs a check. If the result is not within the range, a dictionary entry is reserved for the result of the addition, and a suitable add text entry is inserted to alter the index register immediately before the reference.

Arrays as Parameters

When an array is used as an argument, the location of its first element, P11, is passed in the parameter list. The prologue of the called subroutine contains machine code to compute the corresponding P00 location. When an array has variable dimensions, no constant absorption takes place and the dimension factors are computed for each reference to the array.

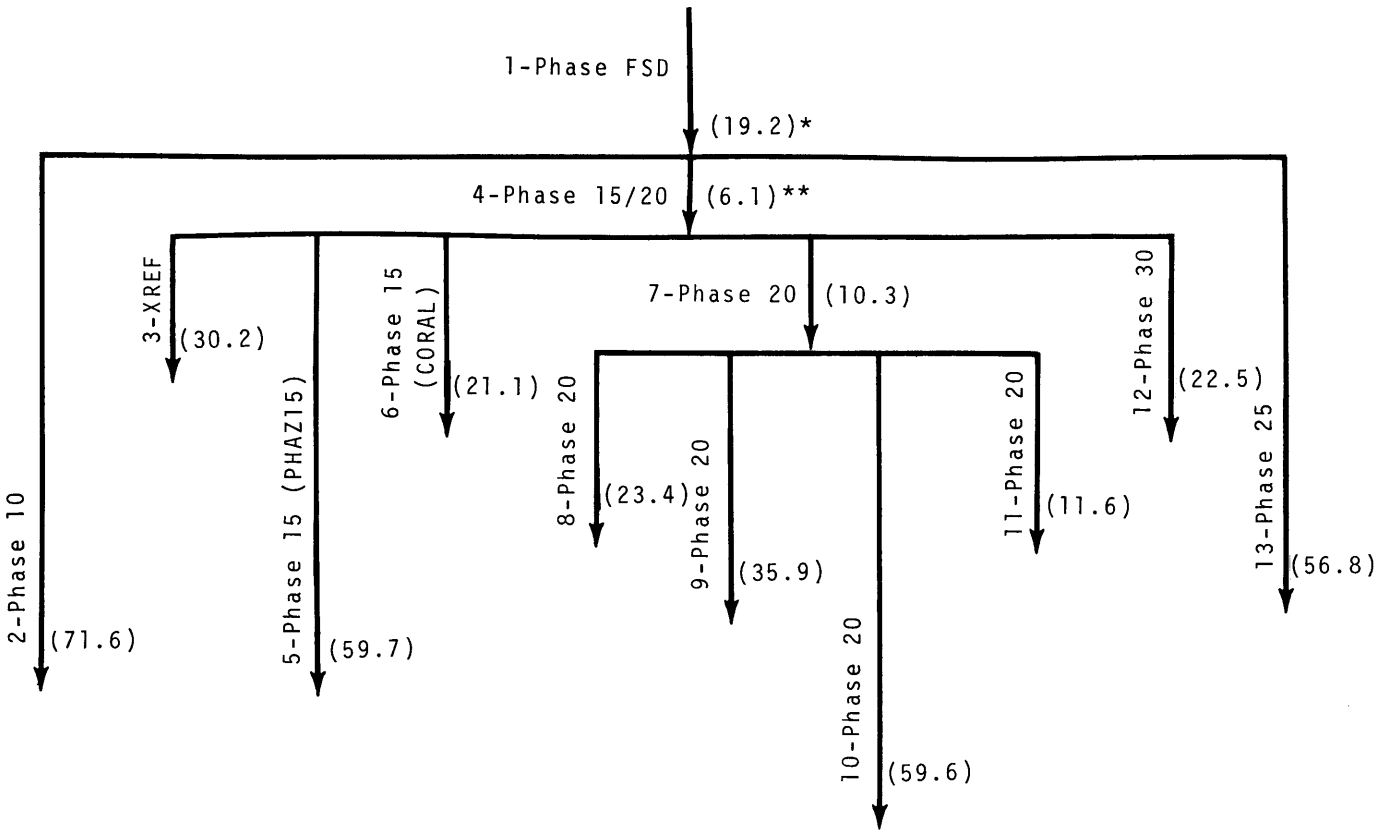
APPENDIX F: COMPILER STRUCTURE

The FORTRAN (H) compiler is structured in a planned overlay fashion. A planned overlay structure is a single load module, created by the linkage editor in response to overlay control statements. These statements, a description of the planned overlay structure, and instructions in specifying such a program structure are presented in the publication IBM System/360 Operating System: Linkage Editor. The processing performed by the linkage editor in response to overlay control statements is described in the publication IBM System/360 Operating System: Linkage Editor, Program Logic Manual.

The compiler's planned overlay structure consists of 13 segments, one of which is the root. The root segment contains the FSD and includes the processing units (e.g., the compile-time input/output routines) and data areas (e.g., communication region) that are used by two or more phases. The root segment remains in main storage throughout the execution of the compiler.

Each of the remaining 12 segments constitutes a phase or a major portion of a phase. Phase segments are overlaid as compiler processing requires the services of another segment.

Figure 55 illustrates the compiler's planned overlay structure. In the illustration, each segment is identified by a number. Segments that originate from the same horizontal line overlay each other as needed. The illustration also indicates the approximate size (in bytes) of each segment.



*The number in parentheses times 1,000 equals the approximate segment length.
**Segment length is dependent upon the SIZE parameter specification.

Figure 55. Compiler Overlay Structure

The longest path[1] of this structure is formed by segments 1, 4, 7, and 10 because, when they are in main storage, the compiler requires approximately 95,200 bytes. Thus, the minimum main storage requirement for the compiler is approximately 105,000 bytes.

The size of segment 4 is dependent upon the value specified for the SIZE parameter of the FORTRAN macro instruction at the time the system is generated. The SIZE parameter affects the amount of storage space required by the CMAJOR (IEKJA2) and RMAJOR (IEKJA4) tables that comprise segment 4. If the default of 204,800 bytes or less is specified, the length of segment 4 is 6.1 as is shown in Figure 55. However, when the SIZE parameter value exceeds the size of the compiler plus its minimum work space (207K), the length of segment 4 increases also.

The linkage editor assigns the relocatable origin of the root segment (the origin of the compiler) at 0. The relocatable origin of each segment is determined by summing the length of all segments in the path. For example, the origin of segment 10 is equal to the length of segment 1 plus the length of segment 4 plus the length of segment 7.

The segments that constitute each phase of the compiler are outlined in Table 32. The remainder of this appendix is devoted to a discussion of the segments of the compiler's planned overlay structure.

Table 32. Phases and Their Segments

Phase	Segment(s) Constituting Phase
Phase 10	Segment 2
XREF	Segment 3
Phase 15	Segments 4, 5, 6
Phase 20	Segments 4, 7, 8, 9, 10, 11
Phase 25	Segment 13
Phase 30	Segment 12
Note: Segment 4 is loaded whenever phases 15, 20, or 30 are loaded. It contains data areas used by 15 and 20.	

Segment 1: This segment is the root segment of the compiler's planned overlay structure. Segment 1 is the FSD. It has a relocatable origin at 0 and is not overlaid by other compiler phases. The composition of segment 1 is illustrated in Table 33.

Segment 2: This segment is phase 10. The origin of the segment is immediately following segment 1. At the completion of phase 10 operation, segment 2 is overlaid by segment 3 if the XREF option was chosen or by segment 4 if the option was not chosen. The composition of segment 2 is illustrated in Table 34.

Table 33. Segment 1 Composition

Control Section	Entry Point(s)
IEKATB	IEKATB
IEKAA01	
IEKAA02	PAGEHEAD
ADCON-IEKAAD	
PUTOUT-IEKAPT	PUTOUT
IEKATM	PHAZSS,PHASB,TST,PHASS, TSP,TOUT,TIMERC
DCLIST-IEKTDC	IEKTDC
AFIXPI-IEKAFP	FIXPI,AFIXPI,FIXPI#
IEKAA00	IEKAGC,ENDFILE,IEKAA9, IEKIORTN
IEKFIOCS	FIOCS#,FIOCS
IEKFCOMH	IBCOM#,IBCOM
IEKTLOAD	IEKUSD,ESD,TXT,IEKTXT, RLD,IEKURL,IEND,IEKUND
ERCOM-IEKAER	
IEKAAA	

Table 34. Segment 2 Composition

Control Section	Entry Point(s)
IEKAINIT	IEKAINIT
STALL-IEKGST	IEKGST
XSUBPG-IEKCSR	IEKCSR
LABTLU-IEKCLT	IEKCLT
XARITH-IEKCAR	IEKCAR
DSPTCH-IEKCDP	IEKCDP,IEKCIN
XIOPST-IEKDIO	IEKDIO
GETCD-IEKCGC	IEKAREAD
CSORN-IEKCCR	IEKCCR,IEKCS3,IEKCS1, IEKCS2,IEKCLC
XTNDED-IEKCTN	IEKCTN
IEKKOS	IEKKOS
XIOOP-IEKCIO	IEKCIO
PUTX-IEKCPX	IEKCPX
XDATYP-IEKCDT	IEKCDT
GETWD-IEKCGW	
XCLASS-IEKDCL	IEKDCL
FORMAT-IEKTFM	IEKTFM
XSPECS-IEKCSP	IEKCSP
XGO-IEKCGO	IEKCGO
XDO-IEKCDO	IEKCDO
PH10-IEKCAA	
IEKXRS	

Segment 3: This segment contains subroutine XREF-IEKXRF. Its origin is immediately following segment 1. If the XREF option is chosen, segment 3 overlays

[1]A path consists of a segment, all segments between it and the root segment, plus the root segment.

Segment 4: This segment is considered a portion of both phases 15 and 20. It contains data areas used by both phases. The origin of segment 4 is immediately following segment 1. Segment 4 is overlaid by segment 13. The composition of segment 4 is illustrated in Table 35.

Table 35. Segment 4 Composition

Control Section	Entry Point(s)
CMAJOR-IEKJA2	
RMAJOR-IEKJA4	

Segment 5: This segment is a portion of phase 15. It contains subroutines that implement the PHAZ15 functions of that phase which are arithmetic translation, text blocking, and information gathering. The origin of segment 5 is immediately following segment 4. Segment 5 is overlaid by segment 6. The composition of segment 5 is illustrated in Table 36.

Table 36. Segment 5 Composition

Control Section	Entry Point(s)
IEKLTB	
LOOKER-IEKLOK	
GENRTN-IEKJGR	IEKJGR
FUNRDY-IEKJFU	IEKJFU
CNSTCV-IEKKCN	IEKKCN
OP1CHK-IEKKOP	IEKKOP,IEKKNG
SUBMULT-IEKKSM	IEKKSM
PHAZ15-IEKJA	IEKJA
BLTNFN-IEKJBF	IEKJBF
STTEST-IEKKST	IEKKST
RELOPS-IEKKRE	IEKKRE
FINISH-IEKJFI	IEKJFI
DFUNCT-IEKJDF	IEKJDF,IEKKPR
MATE-IEKLMA	IEKLMA
ANDOR-IEKJAN	IEKJAN,IEKKNO
CPLTST-IEKJCP	IEKJCP,IEKJMO
UNARY-IEKKUN	IEKKUN,IEKKSW,IEKJEX
DUMP15-IEKLER	IEKLER
PAREN-IEKKPA	IEKKPA
GENER-IEKLGN	IEKLGN
ALTRAN-IEKJAL	IEKJAL
TXTLAB-IEKLAB	IEKLAB
TXTREG-IEKLRG	IEKLRG
SUBADD-IEKKSA	IEKKSA
PH15-IEKJA1	
IEKJA3	

Segment 6: This segment is a portion of phase 15. It contains the subroutines that implement the CORAL functions of the phase. The origin of segment 6 is immediately following segment 4. Segment 6 overlays segment 5 and is overlaid by segment 7. The composition of segment 6 is illustrated in Table 37.

Table 37. Segment 6 Composition

Control Section	Entry Point(s)
DFILE-IEKTDF	IEKTDF
NLIST-IEKTNL	IEKTNL
CORAL-IEKGCR	IEKGCR
NDATA-IEKGDA	IEKGDA
EQVAR-IEKGEV	IEKGEV
CMSIZE-IEKGC2	IEKGCZ
DATOUT-IEKTDT	IEKTDT
IEKGA1	

Segment 7: This segment is a portion of phase 20. It contains the controlling subroutine of that phase, the loop selection routine, and a number of frequently used utility subroutines. The origin of segment 7 is immediately following segment 4. Segment 7 overlays segment 6. The composition of segment 7 is illustrated in Table 38.

Table 38. Segment 7 Composition

Control Section	Entry Point(s)
LPSEL-IEKPLS	IEKPLS
IEKARW	
TARGET-IEKPT	IEKPT
GETDIK-IEKPGK	IEKPGK,IEKPGC,IEKPIV, IEKPFT,IEKPOV
IEKPOP	

Segment 8: This segment is a portion of phase 20. It consists of the subroutines that determine (1) the back dominator, back target, and loop number of each source module block, and (2) the busy-on-exit data. Segment 8 is executed only if the OPT=2 path through phase 20 is followed.

The segment is executed only once and is overlaid by segment 9. The origin of segment 8 is immediately following segment 7. The composition of segment 8 is illustrated in Table 39.

Table 39. Segment 8 Composition

Control Section	Entry Point(s)
SRPRIZ-IEKQAA	IEKQAA,IEKQAB
TOPO-IEKPO	IEKPO
BAKT-IEKPB	IEKPB
BIZX-IEKPZ	IEKPZ
IEKPBL	

Segment 9: This segment is a portion of phase 20. It contains subroutines that perform common expression elimination and strength reduction as well as the major portion of the utility subroutines used during text optimization. Segment 9 is executed only if the OPT=2 path through phase 20 is specified. The origin of segment 9 is immediately following segment 7. During the course of optimization, segment 9 overlays segment 8 and is overlaid by segment 10 after all module loops have been text-optimized. The composition of segment 9 is illustrated in Table 40.

Table 40. Segment 9 Composition

Control Section	Entry Point(s)
KORAN-IEKQKO	IEKQLO
WRITEX-IEKQWT	IEKQWT
CIRCLE-IEKQCL	IEKQCL,IEKQF
PERFOR-IEKQPF	IEKQPF
TYPLOC-IEKQTL	IEKQTL
XSCAN-IEKQXS	IEKQXS,IEKQYS,IEKQZS
XPELIM-IEKQXM	IEKQXM
MOVTEX-IEKQMT	IEKQMT,IEKQDT
CLASIF-IEKQCF	IEKQCF,IEKQPX,IEKQMF
BACMOV-IEKQBM	IEKQBM
REDUCE-IEKQSR	IEKQSR
SUBSUM-IEKQSM	IEKQSM

Segment 10: This segment is a portion of phase 20. It contains full register assignment subroutines, the utility subroutines used by them, and the subroutine that calculates the size of each text block and determines which text blocks can be branched to via RX-format branch instructions. Segment 10 is executed in the optimized paths through phase 20. The origin of segment 10 is immediately following segment 7. The composition of segment 10 is illustrated in Table 41.

Table 41. Segment 10 Composition

Control Section	Entry Point(s)
BLS-IEKSBS	IEKSBS
CXIMAG-IEKRCI	IEKRCI
BKPAS-IEKRBP	IEKRBP
GLOBAS-IEKRGB	IEKRGB
FWDPS1-IEKRF1	IEKRF1
LOC-IEKRL1	
FCLT50-IEKRFL	IEKRFL,IEKRRL,IEKRTF
STXTR-IEKRSX	IEKRSX
FWDPAS-IEKRFP	IEKRFP
SEARCH-IEKRS	IEKRS
REGAS-IEKRRG	IEKRRG
FREE-IEKRFR	IEKRFR
BKDMP-IEKRBK	IEKRBK

Segment 11: This segment is a portion of phase 20. It consists of the subroutines that perform basic register assignment. Segment 11 is executed only in the OPT=0 path through phase 20. The origin of segment 11 is immediately following segment 7. Segment 11 does not overlay any other segment in phase 20, nor is it overlaid by another segment in phase 20. The composition of segment 11 is illustrated in Table 42.

Table 42. Segment 11 Composition

Control Section	Entry Point(s)
SSTAT-IEKRSS	IEKRSS
TALL-IEKRLL	IEKRLL
SPLRA-IEKRSL	IEKRSL

Segment 12: This segment is phase 30. The origin of segment 12 is immediately following segment 4. Segments 4 and 12 overlay segment 13, if errors are encountered during the processing of previous phases. The composition of segment 12 is illustrated in Table 43.

Table 43. Segment 12 Composition

Control Section	Entry Point(s)
MSGWRT-IEKP31	IEKP31
IEKP30-IEKP30	

Segment 13: This segment is phase 25. The origin of segment 13 is immediately following segment 1. Segment 13 overlays segment 4, 7, and either 10 or 11; segment 11 is used for OPT=0 only; segment 10 is used for OPT=1,2 only. The composition of segment 13 is illustrated in Table 44.

Table 44. Segment 13 Composition

Control Section	Entry Point(s)
MAINGN2-IEKVM2	IEKVM2
PACKER-IEKTPK	IEKTPK
LABEL-IEKTLB	IEKTLB
RETURN-IEKTRN	IEKTRN
FNCALL-IEKVFN	IEKVFN
GOTOKK-IEKWKK	IEKWKK
LISTER-IEKTLS	IEKTLS
STOPPR-IEKTSR	IEKTSR
ENTRY-IEKTEN	IEKTEN
CGEN-IEKWCN	
BRLGL-IEKVBL	IEKVBL
IOSUB-IEKTIS	IEKTIS
PROLOG-IEKTPR	IEKTPR
MAINGN-IEKTA	IEKTA
TENTXT-IEKVTN	IEKVTN
IOSUB2-IEKTIO	IEKTIO
END-IEKUEN	IEKUEN
EPILOG-IEKTEP	IEKTEP
IEKGMP	
ADMDGN-IEKVAD	IEKVAD
TSTSET-IEKVTS	IEKVTS
PLSGEN-IEKVPL	IEKVPL
SUBGEN-IEKVSU	IEKVSU
UNRGEN-IEKVUN	IEKVUN
BITNFP-IEKVFP	IEKVFP
FAZ25-IEKP25	

The messages produced by the compiler are explained in the publication IBM System/360 Operating System: FORTRAN IV (G and H) Programmer's Guide. Each message is identified by an associated number. The following table associates a message number with the phase and subroutine in which the corresponding message is generated.

As part of its processing of errors, whenever the compiler encounters an error that is serious enough to cause deletion of a compilation, it prints out: COMPILATION DELETED. (For a more detailed explanation, refer to Appendix D of the aforementioned publication.)

Message Number	Routine in Which Message Number Is Generated	Phase in Which Message Number Is Generated
IEK001I	IEKP30	PHASE 30
IEK002I	XCLASS-IEKDCL	
IEK003I	XARITH-IEKCAR	
IEK005I	XARITH-IEKCAR	
IEK006I	XARITH-IEKCAR, LABTLU-IEKCLT, DSPTCH-IEKCDP, XIOOP-IEKCIO, XCLASS-IEKDCL	
IEK007I	XARITH-IEKCAR	
IEK008I	CSORN-IEKCCR	
IEK009I	CSORN-IEKCCR	
IEK010I	CSORN-IEKCCR	
IEK011I	XARITH-IEKCAR	PHASE 10
IEK012I	CSORN-IEKCCR#	
IEK013I	XARITH-IEKCAR, PUTX-IEKCPX, CSORN-IEKCCR, XCLASS-IEKDCL	
IEK014I	XDATYP-IEKCDT, XSPECS-IEKCSP	
IEK015I	XARITH-IEKCAR	
IEK016I	XGO-IEKCGO	
IEK017I	XGO-IEKCGO	
IEK019I	XGO-IEKCGO	
IEK020I	XGO-IEKCGO	
IEK021I	XGO-IEKCGO	
IEK022I	XGO-IEKCGO	
IEK023I	XTNDED-IEKCTN	
IEK024I	XTNDED-IEKCTN	
IEK025I	XTNDED-IEKCTN	
IEK026I	XTNDED-IEKCTN	
IEK027I	XIOPST-IEKDIO	
IEK028I	XIOPST-IEKDIO	
IEK030I	XDO-IEKCDO	
IEK031I	XDO-IEKCDO	
IEK034I	DSPTCH-IEKCDP	
IEK035I	DSPTCH-IEKCDP	
IEK036I	DSPTCH-IEKCDP	
IEK039I	XTNDED-IEKCTN	PHASE 10
IEK040I	XCLASS-IEKDCL	
IEK047I	XARITH-IEKCAR, XDATYP-IEKCDT	
IEK050I	XARITH-IEKCAR	
IEK052I	DSPTCH-IEKCDP	
IEK053I	XARITH-IEKCAR, DSPTCH-IEKCDP	
IEK056I	XSUBPG-IEKCSR	
IEK057I	XSUBPG-IEKCSR	
IEK058I	XSUBPG-IEKCSR	
IEK059I	XSUBPG-IEKCSR	

Message Number	Routine in Which Message Number Is Generated	Phase in Which Message Number Is Generated
IEK060I	XARITH-IEKCAR, DSPTCH-IEKCDP	
IEK061I	STALL-IEKGST	
IEK062I	XSPECS-IEKCSP STALL-IEKGST	
IEK064I	XTNDED-IEKCTN	
IEK065I	XTNDED-IEKCTN	
IEK066I	XTNDED-IEKCTN	
IEK067I	XTNDED-IEKCTN	
IEK069I	XSPECS-IEKCSP	
IEK070I	XSPECS-IEKCSP	
IEK072I	XSPECS-IEKCSP	
IEK073I	XSPECS-IEKCSP	PHASE 10
IEK074I	XSPECS-IEKCSP	
IEK075I	XSPECS-IEKCSP	
IEK076I	XTNDED-IEKCTN	
IEK077I	XTNDED-IEKCTN	
IEK078I	XTNDED-IEKCTN	
IEK079I	XTNDED-IEKCTN	
IEK080I	XTNDED-IEKCTN	
IEK081I	XTNDED-IEKCTN	
IEK082I	XTNDED-IEKCTN	
IEK083I	XTNDED-IEKCTN	
IEK084I	XTNDED-IEKCTN	
IEK086I	XSPECS-IEKCSP	
IEK087I	XSPECS-IEKCSP	
IEK088I	XSPECS-IEKCSP	
IEK090I	DSPTCH-IEKCDP	
IEK091I	DSPTCH-IEKCDP	
IEK092I	XDATYP-IEKCDT	
IEK093I	XDATYP-IEKCDT	
IEK094I	XDATYP-IEKCDT	
IEK095I	XDATYP-IEKCDT	
IEK096I	XDATYP-IEKCDT	
IEK097I	XTNDED-IEKCTN	
IEK098I	XTNDED-IEKCTN	
IEK099I	XTNDED-IEKCTN	
IEK100I	XTNDED-IEKCTN	
IEK101I	XDO-IEKCDO	
IEK102I	XIOPST-IEKDIO	
IEK104I	XIOPST-IEKDIO	
IEK109I	XIOPST-IEKDIO	
IEK110I	XIOPST-IEKDIO	
IEK111I	XIOPST-IEKDIO	
IEK112I	XGO-IEKCGO, XSPECS-IEKCSP	PHASE 10
IEK113I	XIOPST-IEKDIO	
IEK115I	XIOPST-IEKDIO	
IEK116I	XDO-IEKCDO	
IEK117I	DSPTCH-IEKCDP	
IEK120I	DSPTCH-IEKCDP	
IEK121I	XDATYP-IEKCDT	
IEK122I	XDATYP-IEKCDT	
IEK123I	XDATYP-IEKCDT	
IEK124I	XDATYP-IEKCDT	
IEK125I	XDATYP-IEKCDT	
IEK129I	XDATYP-IEKCDT	
IEK132I	XDATYP-IEKCDT	
IEK133I	XDO-IEKCDO	
IEK134I	XDO-IEKCDO	
IEK135I	XDO-IEKCDO	
IEK136I	XDO-IEKCDO	
IEK137I	XDO-IEKCDO	
IEK138I	XDO-IEKCDO	

Message Number	Routine in Which Message Number Is Generated	Phase in Which Message Number Is Generated
IEK139I	DSPTCH-IEKCDP, XSPECS-IEKCSP, XDATYP-IEKCDT, XTNDED-IEKCTN	
IEK140I	DSPTCH-IEKCDP, XIOPST-IEKDIO	
IEK141I	XIOPST-IEKDIO	
IEK143I	DSPTCH-IEKCDP	
IEK144I	DSPTCH-IEKCDP	
IEK145I	DSPTCH-IEKCDP	
IEK146I	DSPTCH-IEKCDP	
IEK147I	DSPTCH-IEKCDP	
IEK148I	XSPECS-IEKCSP	
IEK149I	XIOPST-IEKDIO	
IEK150I	XSPECS-IEKCSP	
IEK151I	XSPECS-IEKCSP	
IEK152I	XSUBPG-IEKCSR	PHASE 10
IEK153I	XARITH-IEKCAR	
IEK156I	XIOOP-IEKCIO	
IEK157I	XARITH-IEKCAR	
IEK158I	XDO-IEKCDO	
IEK159I	XIOPST-IEKDIO	
IEK160I	XIOOP-IEKCIO, XDO-IEKCDO	
IEK161I	XIOOP-IEKCIO	
IEK163I	XDO-IEKCDO, XARITH-IEKCAR	
IEK164I	XARITH-IEKCAR, XDO-IEKCDO, XIOOP-IEKCIO	
IEK165I	XIOOP-IEKCIO	
IEK166I	XIOOP-IEKCIO	
IEK167I	XARITH-IEKCAR, XSPECS-IEKCSP, XIOPST-IEKDIO, DSPTCH-IEKCDP, XSUBPG-IEKCSR, XDO-IEKCDO	
IEK168I	XSUBPG-IEKCSR	
IEK169I	XIOOP-IEKCIO	
IEK170I	XIOOP-IEKCIO	
IEK171I	XSUBPG-IEKCSR	
IEK176I	XDO-IEKCDO	
IEK192I	XGO-IEKCGO, XCLASS-IEKDCL	
IEK193I	XCLASS-IEKDCL	
IEK194I	XDATYP-IEKCDT	
IEK197I	XIOPST-IEKDIO	
IEK199I	XSUBPG-IEKCSR	
IEK200I	XARITH-IEKCAR	
IEK202I	XDATYP-IEKCDT, XSPECS-IEKCSP	
IEK203I	DSPTCH-IEKCDP	
IEK204I	XIOPST-IEKDIO	
IEK205I	XGO-IEKCGO	
IEK206I	XARITH-IEKCAR	PHASE 10
IEK207I	DSPTCH-IEKCDP	
IEK208I	DSPTCH-IEKCDP	
IEK209I	XDATYP-IEKCDT	
IEK211I	CSORN-IEKCCR	
IEK212I	XIOPST-IEKDIO	
IEK224I	XCLASS-IEKDCL, DSPTCH-IEKCDP	
IEK225I	DSPTCH-IEKCDP	
IEK226I	CSORN-IEKCCR	
IEK229I	XARITH-IEKCAR	
IEK302I	STALL-IEKGST	
IEK303I	STALL-IEKGST	
IEK304I	STALL-IEKGST	PHASE 10 (STALL-IEKGST) and PHASE 15 (CORAL)
IEK306I	STALL-IEKGST	
IEK307I	CORAL-IEKGCR	

Message Number	Routine in Which Message Number Is Generated	Phase in Which Message Number Is Generated
IEK308I	STALL-IEKGST	
IEK310I	STALL-IEKGST	
IEK312I	STALL-IEKGST	PHASE 10 (STALL-IEKGST) and PHASE 15 (CORAL)
IEK314I	STALL-IEKGST	
IEK315I	STALL-IEKGST	
IEK317I	STALL-IEKGST	
IEK318I	NDATA-IEKGDA	
IEK319I	NDATA-IEKGDA	
IEK320I	NDATA-IEKGDA	
IEK322I	STALL-IEKGST	
IEK323I	STALL-IEKGST	
IEK332I	STALL-IEKGST	
IEK334I	STALL-IEKGST	
IEK350I	NDATA-IEKGDA	
IEK352I	NDATA-IEKGDA	
IEK353I	CORAL-IEKGCR	
IEK355I	CMSIZE-IEKGCZ	
IEK356I	STALL-IEKGST	
IEK402I	IEKFIOCS	
IEK403I	IEKFIOCS	FSD
IEK404I	IEKFIOCS	
IEK410I	IEKAINIT	
IEK500I	BLTNFN-IEKJBF DFUNCT-IEKJDF	
IEK501I	DFUNCT-IEKJDF, UNARY-IEKKUN (EXPON)	
IEK502I	UNARY-IEKKUN (EXPON)	PHASE 15 (PHAZ15)
IEK503I	ALTRAN-IEKJAL	
IEK504I	UNARY-IEKKUN	
IEK505I	PHAZ15-IEKJA	
IEK506I	ALTRAN-IEKJAL	

Message Number	Routine in Which Message Number Is Generated	Phase in Which Message Number Is Generated
IEK507I	BLTNFN-IEKJBF	
IEK508I	BLTNFN-IEKJBF	
IEK509I	PHAZ15-IEKJA	
IEK510I	ANDOR-IEKJAN	
IEK512I	FINISH-IEKJFI	
IEK515I	RELOPS-IEKKRE	
IEK516I	FINISH-IEKJFI	
IEK520I	ALTRAN-IEKJAL	PHASE 15 (PHAZ15)
IEK521I	ALTRAN-IEKJAL	
IEK522I	ALTRAN-IEKJAL	
IEK523I	ALTRAN-IEKJAL	
IEK524I	ALTRAN-IEKJAL	
IEK525I	ALTRAN-IEKJAL RELOPS-IEKKRE	
IEK529I	DFUNCT-IEKJDF (IEKKPR)	
IEK530I	SUBADD-IEKKSA	
IEK531I	ALTRAN-IEKJAL	
IEK541I	DFUNCT-IEKJDF	
IEK542I	ALTRAN-IEKJAL	
IEK550I	ALTRAN-IEKJAL, DFUNCT-IEKJDF (IEKKPR)	
IEK552I	DFUNCT-IEKJDF	
IEK570I	GENER-IEKLGN, TXTLAB-IEKLAB, TXTREG-IEKLRG	
IEK580I	ALTRAN-IEKJAL, SUBMLT-IEKKSM, PHAZ15-IEKJA, MATE-IEKLMA, FINISH-IEKJFI	
IEK600I	TOPO-IEKPO	PHASE 20
IEK610I	TOPO-IEKPO	
IEK620I	TOPO-IEKPO	

Message Number	Routine in Which Message Number Is Generated	Phase in Which Message Number Is Generated
IEK630I	TOPO-IEKPO	PHASE 20
IEK640I	GETDIK-IEKPGK	
IEK650I	GETDIK-IEKPGK	
IEK660I	RELCOR-IEKRFL	
IEK661I	FREE-IEKRFR	
IEK662I	FWDPS1-IEKRF1	
IEK670I	BAKT-IEKPB	
IEK671I	BIZX-IEKPZ	
IEK710I	IEKTFM	PHASE 10
IEK720I	IEKTFM	
IEK730I	IEKTFM	
IEK740I	IEKTFM	
IEK750I	IEKTFM	
IEK760I	IEKTFM	
IEK770I	IEKTFM	
IEK800I	MAINGN-IEKTA, TSTSET-IEKVTS, ADMDGN-IEKVAD	PHASE 25
IEK1000I	IEKP30	PHASE 30

APPENDIX H: THE TRACE AND DUMP FACILITIES

Included in the FORTRAN IV (H) compiler are two optional facilities which provide output that can be used to analyze compiler operation and to diagnose compiler malfunction. These two facilities are TRACE and DUMP.

TRACE

The TRACE facility can be used to trace the creation of and the modifications made to the information table and intermediate text, and to provide various other types of diagnostic information. This facility is activated by the inclusion of the TRACE keyword parameter in the PARM field of the EXEC statement used to invoke the compiler. The format of this parameter is:

TRACE=value

where:

value may be either: (1) any one of the basic keyword values that appear in Table 45, or (2) any value that is formed by adding two or more of these basic keyword values.

The type of diagnostic information to be provided by the compiler for a given compilation or batch of compilations is determined according to the value specified for the TRACE keyword. Table 45 defines the type of diagnostic information produced for each of the basic keyword values for the TRACE keyword. If one of these values is specified, the corresponding information is provided by the compiler. For example, if the basic keyword value of 4 is specified, the compiler generates PHAZ15 diagnostic information.

If the value given to the TRACE keyword is the sum of two or more basic keyword values, then the compiler will produce the type of information that corresponds to each basic keyword value that was added to form that value. For example, if the value 20 (the sum of basic keyword values 4 and 16) is specified, the compiler will generate both PHAZ15 diagnostic information and Phase 20 diagnostic information.

Table 45. Basic TRACE Keyword Values and Output Produced

Basic Keyword Values	Output Produced
1	Phase 10 diagnostic information
4	PHAZ15 diagnostic information
16	Phase 20 diagnostic information
64	Printout of: 1. Information table and intermediate text as they appear before the execution of STALL in Phase 10. 2. Information table as it appears after the execution of STALL in Phase 10. 3. Intermediate text as it appears after the execution of PHAZ15 in Phase 15. 4. Information table as it appears after the execution of CORAL in Phase 15. 5. Information table and intermediate text as it appears after the execution of Phase 20.
128	Block size information for each text block (Phase 20)
256	Diagnostic information from the register assignment routines (Phase 20)
512	Diagnostic information from the text optimization routines (Phase 20)
1024	Busy-on-exit information for each text block (Phase 20)
2048	Additional diagnostic information from the register assignment routines (Phase 20)
4096	Printout of intermediate text and information table before and after the execution of Phase 20

DUMP

The dump facility, if activated, will cause abnormal termination of compiler processing if a program interrupt occurs during compilation. It will also cause the main storage areas occupied by the compiler, as well as any associated data and system control blocks to be recorded on an external storage device. The dump facility is activated by including in the compile step of the job: (1) the word DUMP as a parameter in the PARM field of the EXEC statement, and (2) a SYSABEND data definition (DD) statement.

Note: If the DUMP parameter is specified but the SYSABEND DD statement is omitted, abnormal termination, accompanied by an indicative dump, will occur if a program interrupt is encountered. If a program interrupt occurs and the DUMP parameter is not specified, the current compilation will be deleted and the next compilation will be attempted.

APPENDIX I: FACILITIES USED BY THE COMPILER

The following statement, built-in functions and bit-setting facilities are used by the compiler to produce more efficient object code and more efficient use of storage when compiling the compiler. To invoke those routines within the compiler which implement the facilities requires the inclusion of an additional option to the compiler. The option as specified below is coded:

```
PARM.procstep=(...,XL,...)
```

(Note: The XL subparameter is not positional.)

Failure to pass the XL option to the compiler will result in its failure to process these features as documented below. The STRUCTURE statement will be unrecognized and the remaining extensions will be considered as external functions.

STRUCTURE STATEMENT

GENERAL FORM
STRUCTURE//V_{11},V_{12},V_{13},...//V_{21},V_{22},V_{23},...//V_{n1},,V_{n2},V_{n3},...V_n n WHERE: V_{11},V_{12},V_{13},...V_{21},V_{22},V_{23},...V_n n represent names of variables that will be equated to displacement values. If these variables are declared in a Type statement, this statement must precede the STRUCTURE statement.
Note: The // immediately following the word STRUCTURE may be omitted.

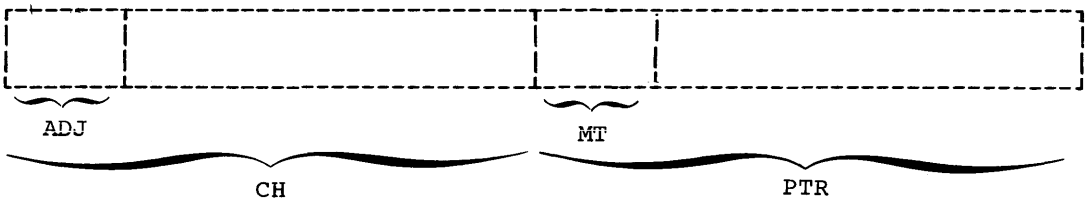
The variables may be implicitly or explicitly declared as any type or length. They must not be dimensioned and must not appear in COMMON or EQUIVALENCE statements. A variable may appear more than once in STRUCTURE statements within a single program or subprogram provided it is given the same displacement by each program.

If D is the name of a structured variable, it must always appear in an executable statement with a single subscript, e.g., D(I). An expression such as D(I) refers to a variable of the type specified for D which is located in main storage at the base address specified by the value of the subscript expression, I, plus a displacement equal to the total number of bytes in the length specification of all the variables preceding D in the STRUCTURE statement in which it appears. For the object program to execute successfully, it is essential that the value of the subscript plus the displacement always be an integral multiple of the length of the referenced field. Displacements may not exceed 255. The subscript expression must be declared as integer or logical.

EXAMPLE:

```
LOGICAL*1       ADJ, MT
INTEGER         CH, PTR
STRUCTURE       CH, PTR//ADJ//CH, MT
```

Here the STRUCTURE statement is used to define a 2-word structure where the high-order byte of each word is overlapped by a 1-byte field.



If J contains a pointer to such a structure, its fields may be referenced as ADJ(J), CH(J), MT(J), and PTR(J).

If a structured variable is used incorrectly the compiler may issue a diagnostic message. A complete list of the FORTRAN IV (H) compiler messages appears in the publication IBM System/360 Operating System: Messages and Codes, Form C28-6631.

BUILT-IN FUNCTIONS

LAND

GENERAL FORM
...=...LAND(a,b)...
WHERE: a, b may be any 1-byte, 2-byte, or 4-byte logical or integer expression.

The value of LAND is obtained by adding the individual bits of the arguments. The resulting value will be considered to be Logical*4 but may be used as an integer.

LOR

GENERAL FORM
...=...LOR(a, b)...
WHERE: a, b may be any 1-byte, 2-byte, or 4-byte logical or integer expression.

The value of LOR is obtained by oring the individual bits of the arguments. The resulting value will be considered to be Logical*4 but may be used as an integer.

LXOR

GENERAL FORM
...=...LXOR(a, b)...
WHERE: a, b may be any 1-byte, 2-byte, or 4-byte logical or integer expression.

The value of LXOR is obtained by exclusive oring the individual bits of the arguments. The resulting value will be considered to be Logical*4 but may be used as an integer.

LCOMPL

GENERAL FORM
...=...LCOMPL(a)
WHERE: a may be any 1-byte, 2-byte, or 4-byte logical or integer expression.

The value of LCOMPL is obtained by complementing the individual bits of the argument. The resulting value will be considered to be Logical*4 but may be used as an integer.

SHFTL and SHFTR

GENERAL FORM
...=...SHFTL(J.K)...; ...=...SHFTR(J,K)...
WHERE: J is a 4-byte variable. K is the actual number of bits to be shifted.

The values of SHFTL and SHFTR are obtained by shifting the first argument left or right the number of bits specified by K. The resulting value will be considered to be Logical*4 but may be used as an integer.

TBIT

GENERAL FORM
...TBIT(A,K)...
WHERE: A is any variable 4-bytes or less in length. K is the number assigned to the bit to be tested.

The value of TBIT is .TRUE. or .FALSE. depending on whether bit position K of the variable A is on or off. Bit 0 is the leftmost bit of variable A. The resulting value will be declared as Logical*4.

MOD 24

GENERAL FORM
...=...MOD 24(A)
WHERE: A must be a 4-byte integer variable.

The value of MOD 24 is the same as its argument except that the high-order byte is set to zero. The resulting value will be declared Integer*4.

BIT-SETTING FACILITIES

BITON

GENERAL FORM
V = BITON(V,K)
WHERE: V must be a single variable; it may be subscripted. K is the number assigned to the bit to be set.

This facility sets the bit at position K in the variable V "on." Bit 0 is the leftmost bit of variable V.

BITOFF

GENERAL FORM
V=BITOFF(V,K)
WHERE: V must be a single variable; it may be subscripted. K is the number assigned to the bit to be set.

This facility sets the bit at position K in the variable V "off." Bit 0 is the leftmost bit of variable V.

BITFLP

GENERAL FORM
V=BITFLP(V,K)
WHERE: V must be a single variable; it may be subscripted. K is the number assigned to the bit to be set.

This facility sets the bit at position K in the variable V to its inverse. Bit 0 is the leftmost bit of variable V.

In all of the bit-setting facilities K is restricted to integer values from 0 to 63 ($0 \leq K \leq 63$). If V is subscripted, the value of the subscript must be the same in both uses, to insure that only a single variable is referenced.

The microfiche directory (Table 46) is designed to help find named areas of code in the program listing, which is contained on microfiche cards at installation. Microfiche cards are filed in alphameric order by object module name. If a control section, entry point, or table is to be located on microfiche, find the name in column one and note the associated object module name. You can then find the item on microfiche, via the object module name; for example, object module IEKOBJT1 is on card IEKOBJT1-1.

The other columns provide a description of the item, its phase, its overlay segment, its flowchart ID (where applicable), and its subroutine directory table number.

Table 46. Microfiche Directory (Part 1 of 8)

Symbolic Name	Description	Object Module Name and CSECT Name	Phase	Overlay Segment	Chart ID * - Only Mentioned in Chart	Sub-routine Directory Table Number
ADMDGN-IEKVAD	Code generation routine	IEKVAD	25	13	--	Table 14
AFIXPI	Entry point	IEKAFP	FSD	1	--	Table 6
AFIXPI-IEKAFP	Exponentiation Routine	IEKAFP	FSD	1	--	Table 6
ALTRAN-IEKJAL	Arithmetic translation routine	IEKJAL	15	5	07	Table 9
ANDOR-IEKJAN	Text generation routine for logical operators	IEKJAN	15	5	07*	Table 9
BACMOV-IEKQBM	Text optimization routine	IEKQBM	20	9	12	Table 12
BAKT-IEKPB	Structural determination routine	IEKPB	20	8	10*	Table 12
BITNFP-IEKVFP	Code generation routine	IEKVFP	25	13	--	Table 14
BIZX-IEKPZ	MVX routine	IEKPZ	20	8	10*	Table 12
BKDMP-IEKRBK	TRACE routine for full register assignment	IEKRBK	20	10	--	Table 12
BKPAS-IEKRBP	Local register assignment routine	IEKRBP	20	10	16	Table 12
BLS-IEKSBS	Branching optimization routine	IEKSBS	20	10	10*	Table 12
BLTNFN-IEKJBF	In-line function routine	IEKJBF	15	5	07*	Table 9
BRLGL-IEKVBL	Code generation routine	IEKVBL	25	13	--	Table 14
CGEN-IEKWCN	Array initialization area	IEKWCN	25	13	--	Table 14
CIRCLE-IEKQCL	Utility subroutine	IEKQCL	20	9	--	Table 13
CLASIF-IEKQCF	Utility subroutine	IEKQCF	20	9	--	Table 13

Table 46. Microfiche Directory (Part 2 of 8)

Symbolic Name	Description	Object Module Name and CSECT Name	Phase	Overlay Segment	Chart ID * - Only Mentioned in Chart	Sub-routine Directory Table Number
CMAJOR-IEKJA2	Backward connection table	IEKJA2	15/20	4	--	Table 10
CMSIZE-IEKGCZ	Base and displacement routine	IEKGCZ	15	6	09*	Table 9
CNSTCV-IEKKCN	Constant conversion routine	IEKKCN	15	5	--	Table 9
CORAL-IEKGCR	Control routine for CORAL segment of phase 15.	IEKGCR	15	6	09	Table 9
CPLTST-IEKJCP	Arithmetic triplet routine	IEKJCP	15	5	07*	Table 9
CSORN-IEKCCR	Collection, conversion, and entry placement routine	IEKCCR	10	2	--	Table 8
CXIMAG-IEKRCI	Local register assignment routine	IEKRCI	20	10	--	Table 12
DATOUT-IEKTDT	DATA statement processing routine	IEKTDT	15	6	09*	Table 9
DCLIST-IEKTDC	Listing routine	IEKTDC	FSD	1	--	Table 6
DELTEX-IEKQDT	Entry point	IEKQMT	20	9	--	Table 13
DFILE-IEKTDF	DEFINE FILE statement routine	IEKTDF	15	6	09*	Table 9
DFUNCT-IEKJDF	In-line, external subprogram, and library function routine	IEKJDF	15	5	07*	Table 9
DSPTCH-IEKCDP	Dispatcher, key word, and utility routine	IEKCDP	10	2	03	Table 8
DUMP15-IEKLER	Error recording routine	IEKLER	15	5	--	Table 9
ENDFILE	Entry point	IEKAA00	FSD	1	01	Table 6
END-IEKUEN	Object module completion routine	IEKUEN	25	13	21	Table 14
ENTRY-IEKTEN	Epilogue and prologue generating routine	IEKTEN	25	13	21*	Table 14
EPILOG-IEKTEP	Subprogram epilogue generating routine	IEKTEP	25	13	21*	Table 14
EQVAR-IEKGEV	COMMON and EQUIVALENCE processing routine	IEKGEV	15	6	09*	Table 9
ESD	Entry point	IEKTLOAD	FSD	1	--	Table 6
FAZ25-IEKP25	COMMON data area	IEKP25	25	13	--	Table 14
FCLT50-IEKRFL	Text checking routine	IEKRFL	20	10	--	Table 12
FILTEX-IEKPFT	Entry point	IEKPGK	20	7	--	Table 13

Table 46. Microfiche Directory (Part 3 of 8)

Symbolic Name	Description	Object Module Name and CSECT Name	Phase	Overlay Segment	Chart ID * - Only Mentioned in Chart	Sub-routine Directory Table Number
FINISH-IEKJFI	Statement completion routine	IEKJFI	15	5	07*	Table 9
FIOCS, FIOCS#	Entry points	IEKFIOCS	FSD	1	--	Table 6
FIXPI, FIXPI#	Entry points	IEKAFP	FSD	1	--	Table 6
FNCALL-IEKVEN	Calling sequence generating routine	IEKVFN	25	13	20*	Table 14
FOLLOW-IEKQF	Entry point	IEKQCL	20	9	--	Table 13
FORMAT-IEKTFM	Generates format text for object module	IEKTFM	10	2	--	Table 8
FREE-IEKRFR	Local register assignment routine	IEKRFR	20	10	--	Table 12
FUNRDY-IEKJFU	Implicit library function reference routine	IEKJFU	15	5	--	Table 9
FWDPAS-IEKRFP	Table building routine	IEKRFR	20	10	15	Table 12
FWDPS1-IEKRF1	Local register assignment routine	IEKRF1	20	10	15*	Table 12
GENER-IEKLGN	Text output routine	IEKLGN	15	5	08	Table 9
GENRTN-IEKJGR	Text entry routine	IEKJGR	15	5	07*	Table 9
GETCD-IEKCGC	Preparatory subroutine	IEKCGC	10	2	03*	Table 8
GETDIC-IEKPGC	Entry point	IEKPGK	20	7	--	Table 13
GETDIK-IEKPGK	Utility subroutine	IEKPGK	20	7	--	Table 13
GETWD-IEKCGW	Utility subroutine	IEKCGW	10	2	--	Table 8
GLOBAS-IEKRGB	Global register assignment routine	IEKRGB	20	10	17	Table 12
GOTOKK-IEKWKK	Branching routine	IEKWKK	25	13	--	Table 14
IBCOM, IBCOM#	Entry points	IEKFCOMH	FSD	1	--	Table 6
IEKAA00	Compiler initialization routine	IEKAA00	FSD	1	01	Table 6
IEKAA01	Default options.	IEKAA01	FSD	1	--	Table 6
IEKAA02	DDNAMES for compiler	IEKAA02	FSD	1		Table 6
IEKAA9	Entry point	IEKAA00	FSD	1	01*	Table 6
IEKAGC	Entry point	IEKAA00	FSD	1	02*	Table 6
IEKAINIT	Processes parameters, gets core	IEKAINIT	FSD	2		Table 6

Table 46. Microfiche Directory (Part 4 of 8)

Symbolic Name	Description	Object Module Name and CSECT Name	Phase	Overlay Segment	Chart ID * - Only Mentioned in Chart	Sub-routine Directory Table Number
IEKAREAD	Entry point	IEKCGC	10	2	--	Table 8
IEKARW	Utility subroutine	IEKARW	20	7	--	Table 13
IEKATB	Diagnostic trace routine	IEKATB	FSD	1	--	Table 6
IEKATM	Timing routine	IEKATM	FSD	1	--	Table 6
IEKCIN	Entry point	IEKCDP	10	2	03*	Table 8
IEKCLC	Entry point	IEKCCR	10	2	--	Table 8
IEKCS1, IEKCS2, IEKCS3	Entry points	IEKCCR	10	2	--	Table 8
IEKFCOMH	Formatted compile-time I/O routine	IEKFCOMH	FSD	1	--	Table 6
IEKFIOCS	Interface between compiler, IEKFCOMH and QSAM	IEKFIOCS	FSD	1	--	Table 6
IEKGA1	COMMON data area for CORAL	IEKGA1	15	6	--	Table 10
IEKGMP	Storage map routine	IEKGMP	25	13	20*	Table 14
IEKIORTN	Entry point	IEKAA00	FSD	1	--	Table 6
IEKJA2	Backward connection table	IEKJA2	15/20	4	--	Table 10
IEKJA3	Function information tables	IEKJA3	15	5	--	Table 1
IEKJA4	Forward connection table	IEKJA4	15/20	4	--	Table 10
IEKJEX	Entry point	IEKKUN	15	5	07*	
IEKJMO	Entry point	IEKJCP	15	5	07*	
IEKKNG	Entry point	IEKKOP	15	5	--	
IEKKNO	Entry point	IEKJAN	15	5	07*	
IEKKOS	Coordinate assignment routine	IEKKOS	10	2	04*	Table 8
IEKKPR	Entry point	IEKJDF	15	5	07*	
IEKKSW	Entry point	IEKKUN	15	5	--	
IEKLTB	Function table	IEKLTB	15	5	--	Table 10
IEKPOV	Entry point	IEKPGK	20	7	--	Table 13
IEKP30	Controlling routine	IEKP30	30	12	22	Table 15
IEKQAB	Entry point	IEKQAA	20	8	--	Table 13

Table 46. Microfiche Directory (Part 5 of 8)

Symbolic Name	Description	Object Module Name and CSECT Name	Phase	Overlay Segment	Chart ID * - Only Mentioned in Chart	Sub-routine Directory Table Number
IEKTLOAD	ESD, TXT, RLD, and loader END record building routine	IEKTLOAD	FSD	1	09*	Table 6
IEKTXT	Entry point	IEKTLOAD	FSD	1	--	Table 6
IEKUND	Entry point	IEKTLOAD	FSD	1	--	Table 6
IEKURL	Entry point	IEKTLOAD	FSD	1	--	Table 6
IEKUSD	Entry point	IEKTLOAD	FSD	1	--	Table 6
IEKXRS	Utility routine for XREF	IEKXRS	10	2	--	Table 8
IEND	Entry point	IEKTLOAD	FSD	1	--	Table 6
INVERT-IEKPIV	Entry point	IEKPGK	20	7	--	Table 13
IOSUB-IEKTIS	Calling sequence generating routine	IEKTIS	25	13	20*	Table 14
IOSUB2-IEKTIO	Calling sequence generating routine	IEKTIO	25	13	--	Table 14
KORAN-IEKQKO	Utility subroutine	IEKQKO	20	9	12*	Table 13
LABEL-IEKTLB	Statement number routine	IEKTLB	25	13	20*	Table 14
LABTLU-IEKCLT	Statement number utility routine	IEKCLT	10	2	--	Table 8
LISTER-IEKTLS	Listing routine	IEKTLS	25	13	--	Table 14
LOC-IEKRL1	Register assignment data area	IEKRL1	20	10	--	Table 12
LOOKER-IEKLOK	Subprogram table look up routine	IEKLOK	15	5	07*	Table 9
LORAN-IEKQLO	Entry point	IEKQKO	20	9	12*	Table 13
LPSEL-IEKPLS	Control routine	IEKPLS	20	7	10	Table 12
MAINGN-IEKTA	Control routine	IEKTA	25	13	20	Table 14
MAINGN2-IEKVM2	Control routine	IEKVM2	25	13	--	Table 14
MATE-IEKLMA	MVS, MVF, and MVX routine	IEKLMA	15	5	--	Table 9
MODFIX-IEKQMF	Entry point	IEKQCF	20	9	--	Table 13
MOVTEX-IEKQMT	Utility subroutine	IEKQMT	20	9	--	Table 13

Table 46. Microfiche Directory (Part 6 of 8)

Symbolic Name	Description	Object Module Name and CSECT Name	Phase	Overlay Segment	Chart ID * - Only Mentioned in Chart	Sub-routine Directory Table Number
MSGWRT-IEKP31	Error message writing routine	IEKP31	30	12	22*	Table 15
NDATA-IEKGDA	Data text routine	IEKGDA	15	6	09*	Table 9
OP1CHK-IEKKOP	Operand one routine	IEKKOP	15	5	--	Table 9
NLIST-IEKTNL	NAMELIST statement routine	IEKTNL	15	6	09*	Table 9
PACKER-IEKTPK	TXT record packing routine	IEKTPK	25	13	--	Table 14
PAGEHEAD	Entry point	IEKAA01	FSD	1	--	Table 6
PAREN-IEKKPA	Parenthesis routine	IEKKPA	15	5	07*	Table 9
PARFIX-IEKQPX	Entry point	IEKQCF	20	9	--	Table 13
PERFOR-IEKQPF	Constant routine	IEKQPF	20	9	--	Table 13
PHASB	Entry point	IEKATM	FSD	1	--	Table 6
PHASS	Entry point	IEKATM	FSD	1	--	Table 6
PHAZSS	Entry point	IEKATM	FSD	1	--	Table 6
PHAZ15-IEKJA	Control routine for PHAZ15 segment of phase 15	IEKJA	15	5	06	Table 9
PH10-IEKCAA	COMMON data area	IEKCAA	10	2	--	Table 8
PH15-IEKJA1	COMMON data area	IEKJA1	15	5	--	Table 1
PLSGEN-IEKVPL	Code generation routine	IEKVPL	25	13	--	Table 14
PROLOG-IEKTPR	Subprogram prologue generating routine	IEKTPR	25	13	21*	Table 14
PUTOUT	Entry point	IEKAPT	FSD	1	--	Table 6
PUTOUT-IEKAPT	Service routine	IEKAPT	FSD	1	--	Table 6
PUTX-IEKCPX	Entry placement utility routine	IEKCPX	10	2	--	Table 8
REDUCE-IEKQSR	Strength reduction routine	IEKQSR	20	9	13	Table 12
REGAS-IEKRRG	Full register assignment routine	IEKRRG	20	10	14	Table 12
RELCOR-IEKRRL	Entry point	IEKRFL	20	10	19*	Table 12
RELOPS-IEKKRE	Relational operator routine	IEKKRE	15	5	07*	Table 9
RETURN-IEKTRN	RETURN statement routine	IEKTRN	25	13	20*	Table 14
RLD	Entry point	IEKTLOAD	FSD	1	--	Table 6

Table 46. Microfiche Directory (Part 7 of 8)

Symbolic Name	Description	Object Module Name and CSECT Name	Phase	Overlay Segment	Chart ID * - Only Mentioned in Chart	Sub-routine Directory Table Number
RMAJOR-IEKJA4	Forward connection table	IEKJA4	15/20	4	--	Table 10
SEARCH-IEKRS	Register loading routine	IEKRS	20	10	17*	Table 12
SPLRA-IEKRSL	Basic register assignment routine	IEKRSL	20	11	--	Table 12
SRPRIZ-IEKQAA	Structured source program listing routine	IEKQAA	20	8	--	Table 13
SSTAT-IEKRSS	Status setting routine	IEKRSS	20	11	10*	Table 12
STALL-IEKGST	COMMON and EQUIVALENCE statement processing routine	IEKGST	10	2	04	Table 8
STOPPR-IEKTSR	STOP and PAUSE statement routine	IEKTSR	25	13	--	Table 14
STTEST-IEKKST	Replacement statement routine	IEKKST	15	5	07*	Table 9
STXTR-IEKRSX	Text updating routine	IEKRSX	20	10	18	Table 12
SUBADD-IEKKSA	Subscript computation routine	IEKKSA	15	5	07*	Table 9
SUBGEN-IEKVSU	Code generation routine	IEKVSU	25	13	20*	Table 14
SUBMLT-IEKKSM	Subscript computation routine	IEKKSM	15	5	07*	Table 9
SUBSUM-IEKQSM	Operand and operand value replacement routine	IEKQSM	20	9	--	Table 13
TALL-IEKRLL	Assigns storage for temporaries	IEKRLL	20	11	--	Table 12
TARGET-IEKPT	Loop and back target routine	IEKPT	20	7	10*	Table 12
TENTXT-IEKVTN	Statement number processing and label map routine	IEKVTN	25	13	20*	Table 14
TIMERC	Entry point	IEKATM	FSD	1	--	Table 6
TNSFM-IEKRTF	Entry point	IEKRFL	20	10	--	Table 12
TOPO-IEKPO	Back dominator routine	IEKPO	20	8	10*	Table 12
TOUT	Entry point	IEKATM	FSD	1	--	Table 6
TSP	Entry point	IEKATM	FSD	1	--	Table 6
TST	Entry point	IEKATM	FSD	1	--	Table 6
TSTSET-IEKVTS	Code generation routine	IEKVTS	25	13	--	Table 14
TXT	Entry point	IEKTLOAD	FSD	1	--	Table 6

Table 46. Microfiche Directory (Part 8 of 8)

Symbolic Name	Description	Object Module Name and CSECT Name	Phase	Overlay Segment	Chart ID * - Only Mentioned in Chart	Sub-routine Directory Table Number
TXTLAB-IEKLAB	Statement number processing	IEKLAB	15	5	08*	Table 9
TXTREG-IEKLRG	Standard text processing routine	IEKLRG	15	5	08*	Table 9
TYPLOC-IEKQTL	Strength reduction routine	IEKQTL	20	9	13*	Table 13
UNARY-IEKKUN	Arithmetic triplet and exponentiation operator routine	IEKKUN	15	5	07*	Table 9
UNRGEN-IEKVUN	Code generation routine	IEKVUN	25	13	--	Table 14
WRITEX-IEKQWT	Diagnostic trace printing routine	IEKQWT	20	9	--	Table 13
XARITH-IEKCAR	Arithmetic routine	IEKCAR	10	2	--	Table 8
XCLASS-IEKDCL	Text generation utility routine	IEKDCL	10	2	03*	Table 8
XDATYP-IEKCDT	DATA and TYPE keyword routine	IEKCDT	10	2	--	Table 8
XDO-IEKCDO	DO keyword routine	IEKCDO	10	2	--	Table 8
XGO-IEKCGO	GO TO keyword routine	IEKCGO	10	2	--	Table 8
XIOOP-IEKCIO	Input/output statement routine	IEKCIO	10	2	--	Table 8
XIOPST-IEKDIO	ASSIGN, RETURN, FORMAT, PAUSE, BACKSPACE, REWIND, END FILE, STOP, and END table entry routine	IEKD10	10	2	--	Table 8
XPELIM-IEKQXM	Common expression elimination routine	IEKQXM	20	9	11	Table 12
XREF-IEKXRF	XREF routine	IEKXRF	10	3	--	Table 8
XSCAN-IEKQXS	Local block scan routine	IEKQXS	20	9	--	Table 13
XSPECS-IEKCSP	COMMON, DIMENSION, and EQUIVALENCE table entry routine	IEKCSP	10	2	--	Table 8
XSUBPG-IEKCSR	CALL, SUBROUTINE, ENTRY, and FUNCTION table entry routine	IEKCSR	10	2	--	Table 8
XTNDED-IEKCTN	DEFINE FILE, NAMELIST, IMPLICIT, andSTRUCTURE table entry routine	IEKCTN	10	2	--	Table 8
YSCAN-IEKQYS	Entry point	IEKQXS	20	9	--	Table 13
ZSCAN POINT	Entry point	IEKQXS	20	9	--	Table 13

APPENDIX K: OBJECT-TIME LIBRARY SUBPROGRAMS

This appendix describes the logic of the FORTRAN IV library subprograms. As the compiler examines the user's FORTRAN source statements and translates them into an object module, it recognizes the need for certain operations the library is designed to perform. At the corresponding points in the object module, the compiler inserts calls to the appropriate library subprograms. At linkage edit time, copies of these library subprograms are made part of the load module. Then, at execution time, the library subprograms perform their various functions. The nature of the user's program determines which and how many library subprograms are included in his load module.

LIBRARY FUNCTIONS

The library performs a variety of functions, which are of five general types:

- load module initialization and termination activities
- input/output operations
- error handling
- data conversion
- mathematical and service functions

It is an important library responsibility to form an interface between the load module and the operating system: library subprograms interface with the data management access methods, provide exit routines for the system interrupt handler and abnormal termination processor, and call the supervisor for various services.

COMPOSITION OF THE LIBRARY

The precise composition and size of a user's version of the FORTRAN IV library will depend on what options he chose at system generation time. The actual location of his permanent library copy (the partitioned data set SYS1.FORTLIB) is also dependent on his installation choice.

A few subprograms, commonly thought of as FORTRAN IV library members, and discussed in this appendix, are not actually members of SYS1.FORTLIB. Instead, they reside in the link library, to be loaded if needed by true library routines at execution time.

SYSTEM GENERATION OPTIONS

At system generation time, the user makes several choices which determine the exact makeup of his FORTRAN IV library These concern:

BOUNDARY ALIGNMENT OPTION: If this option is selected, the IHCADJST routine is included (as a member of the link library) When specification interrupts occur, this routine is loaded to attempt correction of object program data misalignment.

EXTENDED ERROR HANDLING OPTION: If this option is selected, expanded versions of some library routines are included. These provide:

- more precise error messages
- in some cases, more extensive library corrective action and continued execution
- the ability for the user to choose his own or the library's corrective action

The library modules affected by this option are listed in Table 47. A user's library will include either one set of modules or the other.

Table 47. Routines Affected by Extended Error Handling Option

Without Extended Error Handling	With Extended Error Handling
IHCFCOMH	IHCECOMH
IHCUOPT*	IHCUOPT*
IHCDIOSE	IHCEDIOS
IHCFIOSH	IHCEFIOS
IHCFINTH	IHCEFNTH
IHCTRCH**	IHCETRCH
--	IHCERRM***
--	IHCFOPT

*The size differs, although not the name.
**With Extended Error Handling, ICHTRCH becomes an entry point in IHCETRCH.
***Without Extended Error Handling, IHCERRM is an entry point in IHCTRCH.

One other module is affected by system generation choice. IHCUATBL, the data set reference table, has both its length and some contents determined at this time.

MODULE SUMMARIES

IHCFCOMH/IHCECOMH
This module (with its CSECT extension IHCCOMH2) handles the load module initialization and termination activities, and sequential and direct access input/output operations. It also contains switches, addresses, and save areas (at constant displacements from its entry point IBCOM#) that are used by other library routines.

IHCNAMEL
This module directs NAMELIST read/write operations (entry point FRDNL# for reads, entry point FWRNL# for writes).

IHCFIOSH/IHCEFIOS
This module interfaces with the basic sequential access methods to do all sequential input/output for the load module. It is called (at entry point FIOCS#) by IHCFCOMH/IHCECOMH and IHCNAMEL to perform user-requested read/write and device manipulation operations, and by other library routines (such as IHCERRM and IHCFDUMP) to write error messages, traceback maps, user-requested dumps, debug information, and so forth.

IHCDIOSE/IHCEDIOS
This module interfaces with the basic direct access methods to do all direct access input/output for the load module. It is called by the compiler-generated code (at entry point DIOCS#) for DEFINE FILE statements, and by IHCFCOMH/IHCECOMH (at entry point IBCENTRY) for READ, WRITE, and FIND.

IHCFCVTH
This module does data conversion required by other library routines. It is called (at entry point ADCON#) for formatted and namelist input/output, and for other library operations (such as traceback) that require EBCDIC output.

IHCIBERH
This module is called by the compiler-generated code (at entry point IBERH#) to terminate load module execution due to source statement error.

IHCTRCH
This module (entry point IHCERRM) is the library error handling routine when extended error handling has not been specified. It is called by other library routines to direct message printing and produce traceback maps.

IHCETRCH
This module produces traceback maps when the extended error handling facility is present. It can be called by the error monitor IHCERRM (at entry point IHCTRCH), or by the compiler-generated code (at entry point ERRTRA) at user request.

IHCERRM
This module is the error monitor when extended error handling has been specified (otherwise, it is an entry point in IHCTRCH). It can be called by other library routines detecting errors (at CSECT name IHCERRM), by IHCFCOMH/IHCECOMH for termination error summary (entry point IHCERRE), and by the compiler-generated code at user request (entry point ERRMON) for handling of user-detected errors. IHCERRM directs its error handling activities according to the entries in the option table, IHCUOPT.

IHCUOPT
This module is the option table. It contains an 8-byte preface. If extended error handling has been specified, it also contains one entry for each library-defined and user-defined error condition. These entries are used by the error monitor IHCERRM to direct its handling of errors.

IHCFOPT
This module satisfies user requests to examine and modify the option table IHCUOPT. It is called at entry points ERRSAV, ERRSTR, and ERRSET by the compiler-generated code.

IHCFINTH/IHCEFNTH
This module handles certain program interrupts. It is called by the system interrupt handler at entry point ARITH#.

IHCADJST
This module, which is included in the link library only if the user requested boundary alignment at system generation time, is loaded by IHCFINTH/IHCEFNTH to attempt correction of data misalignment that has caused a specification interrupt.

IHCSTAE
: This module, which resides in the link library, is the STAE abnormal termination processor. When IHCFCOMH/IHCECOMH receives control (at entry point EXITRTN1) from the system because the load module has been scheduled for abnormal termination, it loads IHCSTAE to attempt completion of outstanding input/output requests before execution ends.

IHCUTABL
: This module is the unit assignment table. It contains information about the user's data set references, and is used by the library input/output routines in their operations.

IHCFDVCH
: This module is called by the compiler-generated code (entry point DVCHK) at user request to determine if a divide check interrupt occurred.

IHCFOVER
: This module is called by the compiler-generated code (entry point OVERFL) at user request to determine whether or not overflow or underflow interrupts occurred.

IHCFSLIT
: This module is called by the compiler-generated code (entry points, SLITE, SLITET) at user request to set or test private switches ("pseudo-sense lights").

IHCFEXIT
: This module is called by the compiler-generated code (entry point EXIT) at user request to terminate load module execution.

IHCFDUMP
: This module is called by the compiler-generated code (entry points DUMP, PDUMP) at user request to produce a dump of specified areas of main storage.

MATHEMATICAL ROUTINES: Information on these library modules can be found in the publication IBM System/360 Operating System: FORTRAN IV Library--Mathematical and Service Subprograms, Order No. GC28-6818.

LIBRARY INTERRELATIONSHIPS

It is helpful to recognize that there is not always a one-to-one relationship between library functions and library modules. Some functions require the execution of several modules, and, conversely, some modules are involved with more than one function.

Certain library modules are called primarily by the compiler-generated code, but a large number are called only by other library modules or by the system. This relationship is illustrated in Figure 56.

In interfacing with each other, with the system and with the compiler-generated code, library modules use nonstandard calling and register-saving procedures.

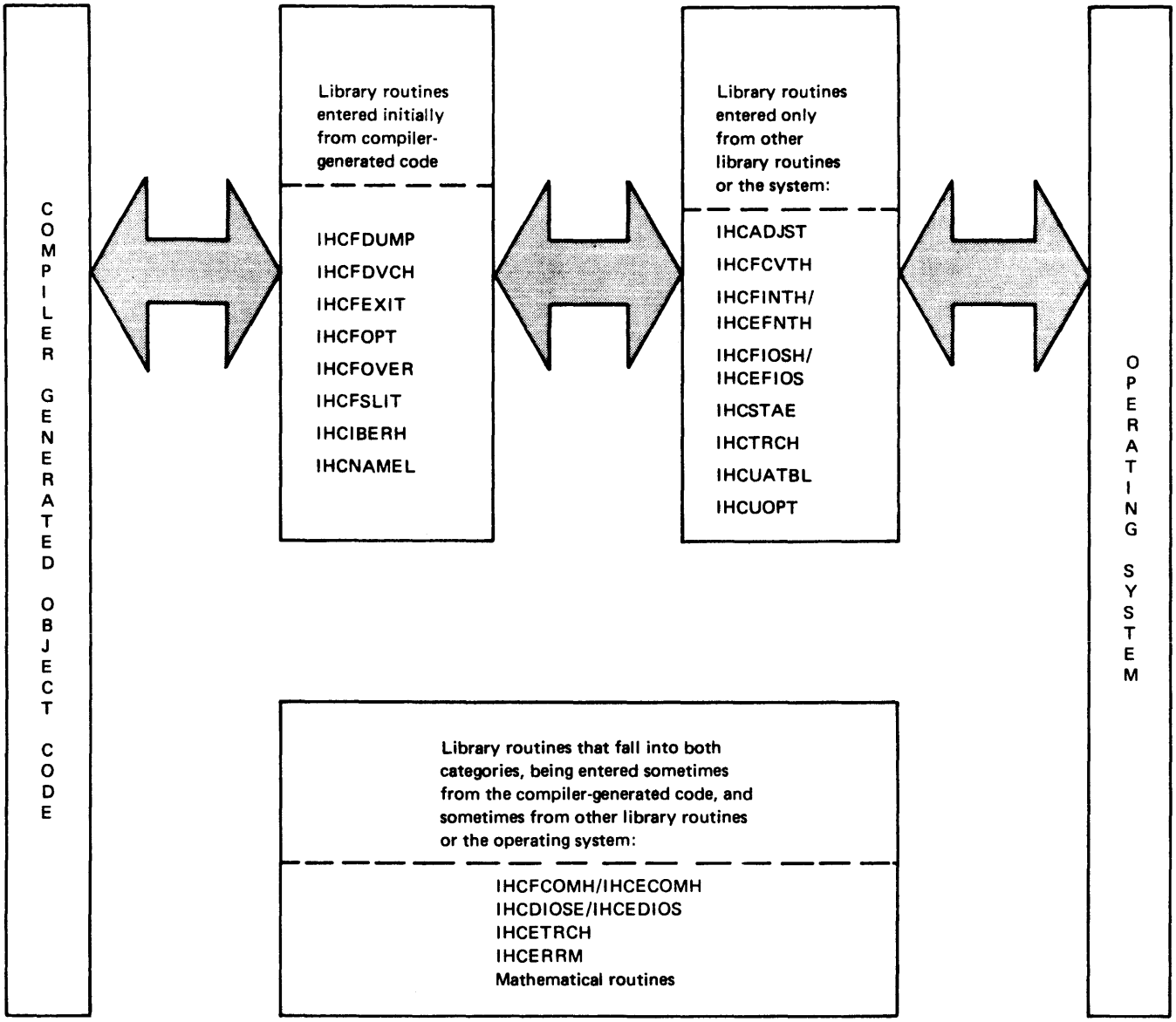


Figure 56. Calling Paths for Library Routines

INITIALIZATION

The library is responsible for the load module's initialization activities. Every compiler-generated main program begins with a branch to the IBFINT section IHCFCOMH/IHCECOMH (see Table 51 for IHCFCOMH/IHCECOMH branches and subroutines). This library routine performs the following initialization procedure:

- Saves the load module entry point in its location MAINEP, and the main program's save area pointer in its location REG13.
- Issues a SPIE macro instruction specifying library control for program interrupts 9, 11, 12, 13, 15, and, if boundary alignment was selected at system generation time, 6.
- Issues a STAE macro instruction specifying library control if the system schedules the load module for abnormal termination.
- Calls IHCFIOSH/IHCEFIOS to open the object error unit.

Control is then returned to the main program, which begins its processing.

INPUT/OUTPUT OPERATIONS

Processing FORTRAN input/output requests is mainly the responsibility of the library. For each request, the compiler sets up a call(s) to the appropriate entry point in the appropriate library routine. For NAMELIST READ/WRITE, the call is to IHCNAMEL, which then calls IHCFIOSH/IHCEFIOS and IHCFCVTH. For DEFINE FILE, the call is to IHCDIOSE/IHCEDIOS. For all other operations, the call is to IHCFCOMH/IHCECOMH (see Table 51 for IHCFCOMH/IHCECOMH branches and subroutines). If the operation is sequential READ/WRITE, the IHCFCOMH/IHCECOMH routine calls IHCFIOSH/IHCEFIOS (and also IHCFCVTH if format control is present). If the operation is REWIND, BACKSPACE, or ENDFILE the IHCFCOMH/IHCECOMH routine calls IHCFIOSH/IHCEFIOS. If the operation is direct access READ, WRITE, or FIND, routine IHCFCOMH/IHCECOMH calls IHCDIOSE/IHCEDIOS (and IHCFCVTH if format control is present). If the operation is STOP with message, or PAUSE, routine IHCFCOMH/IHCECOMH calls the supervisor. This flow is outlined in Figure 57. For each direct access or sequential read/write request, the compiler-generated code issues multiple calls to IHCFCOMH/IHCECOMH: an initial call, one call for each item (either variable or array) in the I/O list and a final call. Thus, the FORTRAN statement READ (23,100)Z,Y,X results in five consecutive calls to IHCFCOMH/IHCECOMH.

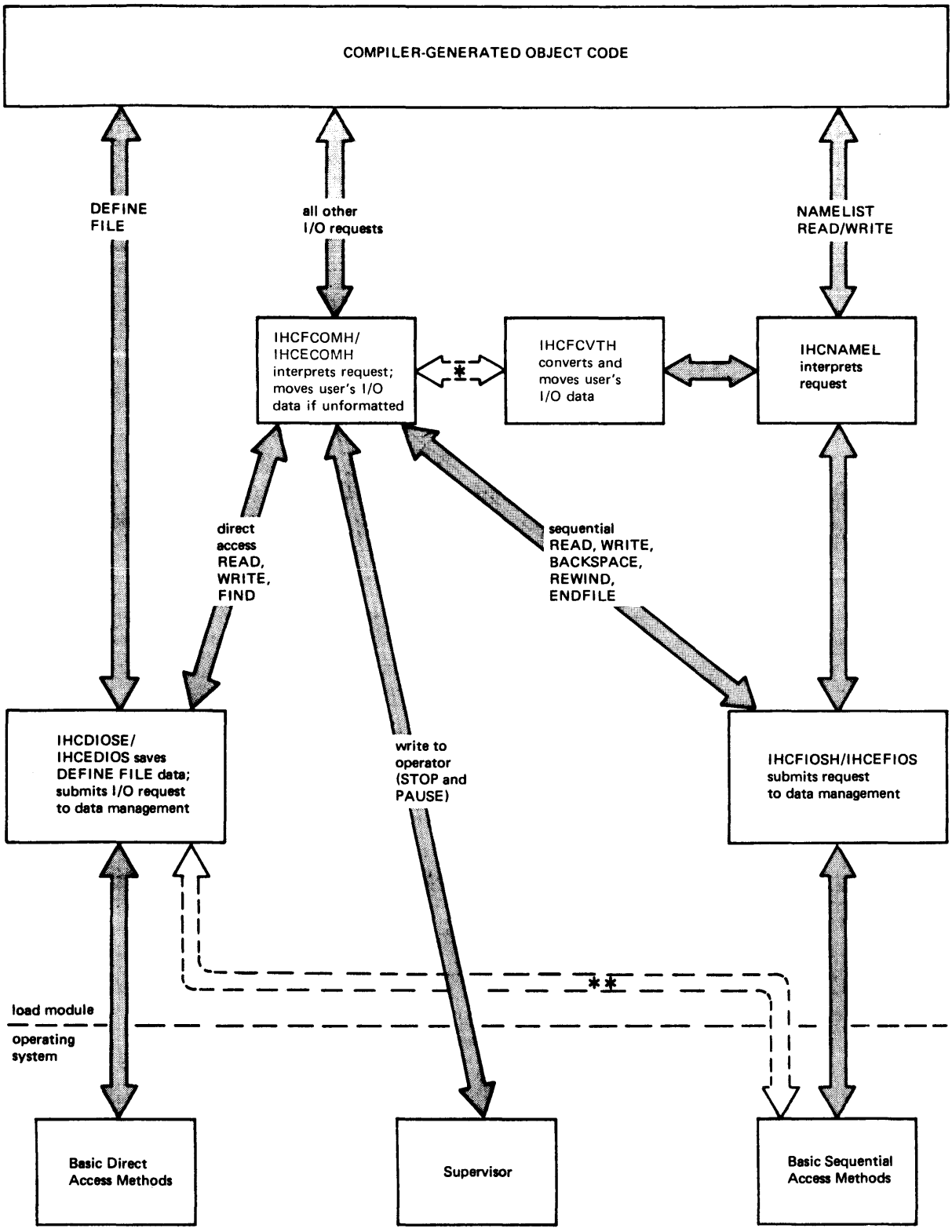


Figure 57. Control Flow for Input/Output Operations

DEFINE FILE

The compiler-generated code branches directly to IHCDIOSE/IHCEDIOS at entry point DIOCS#. This section takes the address of the parameter list containing the data set characteristics supplied by the user and places it in the appropriate unit assignment table (IHCUATBL) entry. There may be more than one data set defined per DEFINE FILE statement, in which case DIOCS# loops through the definitions, placing the parameter list addresses into the table.

If a data set has been previously defined, the new definition is ignored. If the data set requested is sequential rather than direct, IHCERRM is called with error condition 235 indicated. If the data set is the object error unit, IHCERRM is called with error 234 indicated.

DIOCS# also places the address of the section in IHCDIOSE/IHCEDIOS that handles actual reads and writes--IBCENTRY--into a fixed location in IHCFCOMH/IHCECOMH, in order to establish addressability for later branching. If the user fails to place his DEFINE FILE statement ahead of his associated READ or WRITE statement, this address will not be available, and an error condition will occur.

DIOCS# returns to the compiler-generated code.

SEQUENTIAL READ/WRITE WITHOUT FORMAT

Initial Call

The initial call is to IHCFCOMH/IHCECOMH, which saves END= and ERR= addresses, if they are present, in its locations ENDFILE and IOERROR, respectively, and then branches to IHCFIOSH/IHCEFIOS, passing along the data set reference number.

IHCFIOSH/IHCEFIOS uses this data set reference number to consult the corresponding entry in the table IHCUATBL. (This table is explained in Figures 58 and 59.) The initialization action taken by IHCFIOSH/IHCEFIOS depends on the nature of the previous operation performed on this data set. The previous operation possibilities are:

- no previous operation
- previous operation was read or write
- previous operation was backspace
- previous operation was write end of file
- previous operation was rewind

NO PREVIOUS OPERATION: IHCFIOSH/IHCEFIOS must create a unit block, which will contain the DCB, DECBs, and other library information to be used in controlling operations. Space for the unit block is acquired with a GETMAIN, and a pointer to it is stored in the IHCUATBL entry. (The contents of the unit block are outlined in Figure 60.)

IHCFIOSH/IHCEFIOS inserts certain standard values into the DCB in the unit block. It does this by moving in a copy of a nonfunctioning skeleton DCB, which specifies DSORG as PS, MACRF as (R,W), DDNAME as FTnnF001, and gives addresses in IHCFIOSH/IHCEFIOS for SYNAD and EODAD, and for EXLST, which specifies the OPEN exit routine. IHCEFIOSH/IHCEFIOS puts the data set reference number into the nn field of the DDNAME. This establishes for the system the connection between this DCB and the user's DD card, which must have the same name on it.

IHCFIOSH/IHCEFIOS now issues an OPEN macro instruction, which merges the user's DD information, and label information if the data set already exists. When its open exit routine (IHCDCBXE) gains control, IHCFIOSH/IHCEFIOS examines the DCB. If fields are zero, indicating the user has omitted corresponding DD parameters, IHCFIOSH/IHCEFIOS inserts library default values. (These default values are stored in the IHCUATBL entry. See Table 50 for a list of the default values.)

After completion of the OPEN macro, IHCFIOSH/IHCEFIOS places the buffer address(es) in the housekeeping section of the unit block, and also in the DECB(s). It also puts the DCB address into the DECB(s). If this is a read operation, it sets the first byte of the type of input/output request field in the DECB(s) to X'80', indicating the reads should be of blocksize; if this is a write operation, it sets this byte to X'00', indicating the writes should be of logical record length.

If the initialization is for a read operation, IHCFIOSH/IHCEFIOS now issues a READ macro, with a CHECK, filling the buffer. If double buffering is in effect, it also issues a second READ macro, to begin filling the second buffer. (This READ is not checked until IHCFIOSH/IHCEFIOS is entered the next time for this data set.) Control is returned to

IHCFCOMH/IHCECOMH, along with address and length of the data that was read.

If the initialization is for a write operation, IHCFIOSH/IHCEFIOS simply returns to IHCFCOMH/IHCECOMH, passing the address and length of the buffer. (The actual write operation will not take place until IHCFCOMH/IHCECOMH fills the buffer.)

PREVIOUS OPERATION--READ OR WRITE: In this case, the data set is already open and the unit block in existence. The DECB is set to indicate the proper action (either read or write). If this is a write request, control is returned to IHCFCOMH/IHCECOMH with buffer address and length. If it is a read request, the READ macro is issued to fill the buffer, and the address and length of the data that was read is passed back to IHCFCOMH/IHCECOMH.

PREVIOUS OPERATION--BACKSPACE: The operation is the same as for "Previous Operation--Read or Write" described above, except that priming of buffer(s) may be needed.

PREVIOUS OPERATION--END FILE: IHCFIOSH/IHCEFIOS must first close the existing data set, and process a new one. To process a new data set, IHCFIOSH/IHCEFIOS increments the sequence number of the DDNAME field in the old DCB; for example, FT14F001 is changed to FT14F002. The OPEN procedure described above under "No Previous Operation" is then followed. (The library assumes the user has a FTnnF002 DD card for this new data set.) The usual read or write procedure is used.

PREVIOUS OPERATION--REWIND: The data set has been closed, and must be reopened. The procedure is the same as that described under "No Previous Operation," beginning after the creating of the unit block.

In all of the above cases, IHCFIOSH/IHCEFIOS returns to IHCFCOMH/IHCECOMH, which saves the buffer pointer and length, and then returns to the compiler-generated code.

Second Call

The compiler-generated code calls IHCFCOMH/IHCECOMH, passing information about the first item in the I/O list (its address, type, whether it is a variable or array, etc.). If this is a read request for a variable, IHCFCOMH/IHCECOMH takes the proper number of bytes from the buffer and moves them to the indicated address. For an array, IHCFCOMH/IHCECOMH repeats the process, filling the array element by element. If this is a write request for a variable, IHCFCOMH/IHCECOMH takes the item from the indicated address and moves it into the buffer. For an array, IHCFCOMH/IHCECOMH repeats the process, emptying the array element by element. After adjusting its buffer pointer so it points to either the next data item or the next empty space, IHCFCOMH/IHCECOMH returns to the compiler-generated code.

Additional List Item Calls

The procedure is the same as for the first list item, with these exceptions. When IHCFCOMH/IHCECOMH is processing a read request and finds it has emptied the buffer, it calls IHCFIOSH/IHCEFIOS to issue another READ macro and refill it. If double buffering is in effect, IHCFIOSH/IHCEFIOS passes the address of the other buffer (after checking the READ macro for that buffer), and then issues a READ macro instruction for the buffer just emptied, always keeping one READ ahead.

When IHCFCOMH/IHCECOMH is processing a write request and finds it has filled the buffer, it calls IHCFIOSH/IHCEFIOS to issue the actual WRITE macro. If double buffering is in effect, IHCFIOSH/IHCEFIOS passes back the address of the other buffer.

Final Call

For a read operation, the main program passes control to IHCFCOMH/IHCECOMH which passes control on to IHCFIOSH/IHCEFIOS. If IHCFIOSH/IHCEFIOS finds that, for this data set, physical records are larger than logical records, it simply returns to IHCFCOMH/IHCECOMH, which returns to the compiler-generated object code. If physical records are shorter than logical records, IHCFIOSH/IHCEFIOS issues READ macros until it reaches the end of the logical record. This positions the device at the beginning of the next logical record, in preparation for subsequent FORTRAN READ requests for this unit.

For a write operation, IHCFCOMH/IHCECOMH gives control to IHCFIOSH/IHCEFIOS. If the data set is unblocked, or if it is blocked and the buffer is full, IHCFIOSH/IHCEFIOS issues a final WRITE macro.

System Block Modification and Reference

While performing its functions, IHCFIOSH/IHCEFIOS may modify certain fields of the current DCB:

DCBBLKSI--IHCFIOSH/IHCEFIOS changes this field before writing out a short block when RECFM=FB. IHCFIOSH/IHCEFIOS restores it after issuing the corresponding CHECK macro.

DCBOFLGS--before issuing a CLOSE (TYPE=T) macro to implement an ENDFILE request, IHCFIOSH/IHCEFIOS turns on the high order bit to make this look like an output data set.

IHCFIOSH/IHCEFIOS also modifies some fields of the DECB(s), in addition to its initialization:

DECTYPE (byte 1)--for reads, set to indicate a read of blocksize; for writes, set to indicate a write of logical record size.

DECTYPE (byte 2)--set to indicate read or write when the previous operation for this data set was the opposite.

DECLNGTH--filled in when a U-type record is to be written.

In addition to referring to the DCB and DECB(s), IHCFIOSH/IHCEFIOS also examines the CSW field in the Input/Output Block (IOB) to get the residual count. (The DECB points to the IOB.) By subtracting the residual count from the DCB blocksize, IHCFIOSH/IHCEFIOS knows the actual length of the data read into the buffer.

Error Conditions

During their processing of unformatted sequential reads and writes, IHCFIOSH/IHCEFIOS and IHCFCOMH/IHCECOMH check at various times for a number of error conditions. IHCFIOSH/IHCEFIOS checks for the following error conitions: the user's data set reference number is out of IHCUATBL range (error 220); he failed to supply a DD card for the requested data set (error 219); and he specified anything other than Variable Spanned (VS) records (error 214); IHCFCOMH/IHCECOMH checks each I/O list item to see if it exceeds buffer size (error 213). If one of these errors is detected, control is passed to IHCERRM.

If extended error handling is in effect, control returns from IHCERRM to its caller, which does the following:

- conditions 219 or 220 -- IHCEFIOS returns to its original caller at the error displacement. (The error displacement is 2 bytes beyond the address originally passed to it in register 0; the normal return point is 6 bytes beyond the address originally passed in register 0.)
- condition 214 -- if user-supplied corrective action is indicated or if the operation is a read, IHCEFIOS ignores the input/output request and returns to the error displacement. Otherwise, it changes the record format to VS and continues execution.
- condition 213 -- IHCECOMH ignores the list item request, and any further list item requests for this read or write.

If an end-of-file is detected when IHCFIOSH/IHCEFIOS issues a CHECK macro, its EODAD routine gains control. It branches to the user's END= address if one exists. If not, it branches to IHCERRM. Without extended error handling, this is a terminal error. With extended error handling, control returns to IHCEFIOS after error message and traceback printing, and possible user corrective action. IHCEFIOS T-closes the data set, and returns to its original caller at the error displacement.

If an input/output error is detected when IHCFIOSH/IHCEFIOS issues a CHECK macro, its SYNAD routine gains control. It issues a GETMAIN for extra space, and then issues a SYNADAF macro, which puts relevant information into the area. (If extended error handling exists, IHCEFIOS has the associated data set reference number converted and places it into the error message--218.) IHCFIOSH/IHCEFIOS next asks data management to accept the data in error, and restart the IOB chain. IHCERRM is then called. Without extended error handling, the error message and traceback are printed, and then IHCERRM branches to the user's ERR= address if there is one, and to the IBEXIT section of IHCFCOMH if there was not. With extended error handling, IHCERRM goes to the user's option table exit routine if there is one and, in any case, prints out the error message and traceback. Then it branches to the user's ERR= address, if there is one. If not, it returns to IHCEFIOS, which continues processing if the user supplied his own corrective action; if not, IHCEFIOS returns to the error displacement of the routine that originally called it.

SEQUENTIAL READ/WRITE WITH FORMAT

These operations are the same as for sequential read/write without format, except IHCFCOMH/IHCECOMH must scan and interpret the associated format specification, and control the conversion and movement of list items accordingly.

Processing the Format Specification

OPENING SECTION: Upon return from the initialization section of IHCFIOSH/IHCEFIOS, IHCFCOMH/IHCECOMH begins examining the format specification, the address of which is passed as an argument in the initial branch from the compiler-generated code. The format specification may be one of two types: one declared in a FORMAT statement in the FORTRAN source program; or an array that the user has filled in with format information during execution (often referred to as object-time format specification). In the former case, the compiler has already translated the statement into an internal code. In the latter, the format information exists in its EBCDIC form, just as it would in a FORMAT statement.

In the case of an object-time format specification, IHCFCOMH/IHCECOMH must pick up the array contents and process them so they are in the same form as a format specification processed by the compiler. IHCFCOMH/IHCECOMH does this using the TRT instruction and its table TRTSTB.

The translated format codes, and their meanings to IHCFCOMH/IHCECOMH, are listed in Table 48.

In both cases, IHCFCOMH/IHCECOMH now begins scanning the format information. It reads it -- saving the control information -- until it finds the first conversion code (or the end of the FORMAT statement). Then it exits to the compiler-generated code.

LIST ITEM CALLS FOR READ REQUEST: When IHCFCOMH/IHCECOMH is entered for the first list item, it determines from the conversion code which section of the conversion routine IHCFCVTH to call. It passes information from the format specification, (such as scale and width), information about the list item (such as its address), and buffer address and length. IHCFCVTH, and its associated subroutines, do both the conversion and the moving of the data from buffer to list item location or vice versa.

In general, after a conversion routine has processed a list item, IHCFCOMH/IHCECOMH determines whether or not that routine can be applied to the next list variable or array element (if an array is being processed). IHCFCOMH/IHCECOMH examines a field count in the format specification that indicates the number of times a particular conversion code is to be applied to successive list variables or elements of an array.

If the conversion code is to be repeated and if the previous list item was a variable, IHCFCOMH/IHCECOMH returns control to the main program. The main program again branches to IHCFCOMH/IHCECOMH and passes, as an argument, the main storage address assigned to the next list item.

Table 48. Format Code Translations and Their Meanings (Part 1 of 2)

Source FORMAT Code	Code After Compiler or IHCFCOMH/ IHCECOMH Translation (in hex)	Description	Type	Corresponding Action by IHCFCOMH/IHCECOMH
beginning of statement			control	Save location for possible repetition of the format codes; clear counters.
n(	04n	group count (n=1-byte value of repeat count; set to 1 if no repeat count)	control	Save n and location of left parenthesis for possible repetition of the format codes in the group.
a	06a	field count (a=1-byte value of repeat count)	control	Save a for repetition of format code that follows.
nP	08*	scaling factor	control	Save n for use by F, E, and D conversions.
Tn	12n	column reset (n=1-byte value)	control	Reset current position within record nth column or byte.
nX	18n	skip or blank (n=1-byte value)	control	Skip n characters of an input record, or insert n blanks in an output record.
'text' or nH	1Aw	literal data	control	Move w characters from an input record to the FORMAT statement, or w characters from the FORMAT statement to an output record.

Notes: * is a 1-byte value of n, if n was positive; if negative, it is the value plus 128(decimal).
w = 1-byte value of field width.
d = 1-byte value of number of digits after the decimal point.

Table 48. Format Code Translations and Their Meanings (Part 2 of 2)

Source FORMAT Code	Code After Compiler or IHCFCOMH/ IHCECOMH Translation (in hex)	Description	Type	Corresponding Action by IHCFCOMH/IHCECOMH
Fw.d	OA w.d	F-conversion	conversion	IHCFCOMH/IHCECOMH passes the values of w, d and p --plus information about the list item and the buffer-- to the appropriate section of IHCFCVTH for conversion.
Ew.d	OC w.d	E-conversion	conversion	
Dw.d	OE w.d	D-conversion	conversion	
Iw	10 w	I-conversion	conversion	
Aw	14 w	A-conversion	conversion	
Gw.d	20 w.d	G-conversion	conversion	
Lw	16 w	L-conversion	conversion	
Zw	24 w	Z-conversion	conversion	
)	1C	group end	control	Test group count. If it is greater than 1, repeat format codes in group; otherwise, continue to process FORMAT statement from current position.
	1E	record end	control	Input or output one record using IHCFIOSH/IHCEFIOS/ and READ/WRITE macro instruction.
		end of statement	control	If no I/O list items remain to be transmitted, return control to load module to link to the closing section; if I/O list items remain, read or write one record using input/output interface and the READ/WRITE macro instruction. Repeat format codes from last parenthesis.

Notes: * is a 1-byte value of n, if n was positive; if negative, it is the value plus 128(decimal).
w = 1-byte value of field width.
d = 1-byte value of number of digits after the decimal point.

If the conversion code is to be repeated and if an array is being processed, IHCFCOMH/IHCECOMH computes the main storage address of the next element in the array. The conversion routine that processed the previous element is then given control. This procedure is repeated until either all the array elements associated with a specific conversion code are processed or end of logical record is detected (error 212). In the latter case, control is passed to IHCERRM.

If the conversion code is not to be repeated, control is passed to the scan portion of IHCFCOMH/IHCECOMH to continue the scan of the format specification. If the scan portion determines that a group of conversion codes is to be repeated, the conversion routines corresponding to those codes are applied to the next portion of the input data. This procedure is repeated until either the group count is exhausted or the input data for the READ statement is exhausted.

If a group of conversion codes is not to be repeated and if the end of the format specification is not encountered, the next format code is obtained. For a control type code, control is passed to the associated control routine to perform the indicated operation. For a conversion type code, control is returned to the compiler-generated code if the previous list item was a variable. The compiler-generated code again branches to IHCFCOMH/IHCECOMH and passes, as an argument, the main storage address assigned to the next list item. Control is then passed to the conversion routine associated with the new conversion code. The conversion routine then processes the data for this list item. If the data that was

just converted was placed into an element of an array and if the entire array has not been filled, IHCFCOMH/IHCECOMH computes the main storage address of the next element in the array and passes control to the conversion routine associated with the new conversion code. The conversion routine then processes the data for this array element.

If, in the midst of its processing, IHCFCOMH/IHCECOMH finds that it has emptied the buffer it calls IHCFIOSH/IHCEFIOS to issue another READ macro instruction.

If the scan portion encounters the end of the format specification and if all the list items are satisfied, control returns to the next sequential instruction within the compiler-generated code. This instruction (part of the calling sequence to IHCFCOMH/IHCECOMH) branches to the closing section. If all the list items are not satisfied, control is passed to the input/output interface to read (via the READ macro instruction) the next input record.

LIST ITEM CALLS FOR WRITE REQUEST: IHCFCOMH/IHCECOMH processing is similar to that for a read request. The main difference is that the conversion routines obtain data from the main storage addresses assigned to the list items rather than from an input buffer. The converted data is them transferred to an output buffer. If all the list items have not been converted and transferred prior to the encounter the end of the format specification, control is passed to the IHCFIOSH/IHCEFIOS. The IHCFIOSH/IHCEFIOS writes (via the WRITE macro instruction) the contents of the current output buffer onto the output data set.

Formatting control for the remaining list items is then resumed at the group count of the left parenthesis corresponding to the last preceding right parenthesis, or, if none exists, from the first left parenthesis.

If IHCFCOMH/IHCECOMH detects an error in the format specification (condition 211), it calls IHCERRM. Standard corrective action in the case of extended error handling is to treat the invalid character as a terminal right parenthesis and continue execution.

CLOSING SECTION: If the operation is a read request, the closing section simply returns control to the main program to continue execution. If the operation is a write requiring a format, the closing section branches to the IHCFIOSH/IHCEFIOS. The IHCFIOSH/IHCEFIOS writes (via the WRITE macro instruction) the contents of the current input/output buffer (the final record) onto the output data set. IHCFIOSH/IHCEFIOS then returns control to the closing section. The closing section, in turn, returns control to the compiler-generated code.

DIRECT ACCESS READ/WRITE WITHOUT FORMAT

Unformatted reading and writing for direct access data sets is handled by IHCFCOMH/IHCECOMH and IHCDIOSE/IHCEDIOS. The procedure is similar to that for sequential data sets. The compiler-generated object code calls IHCFCOMH/IHCECOMH once for initialization, once for closing, and once in between for each item (variable or array) in the I/O list. IHCFCOMH/IHCECOMH calls IHCDIOSE/IHCEDIOS once for initialization, once for closing (if it is a write request), and as many times in between as the input/output data requires. The actions IHCFCOMH/IHCECOMH are identical to those for sequential unformatted read and write operations. The only exception is that IHCDIOSH/IHCEDIOS is called in place of IHCFIOSH/IHCEFIOS.

Initialization Branch

When IHCDIOSE/IHCEDIOS is given control, it checks the entry in IHCUATBL corresponding to the indicated data set reference number to see if the data set has been opened. If not, IHCDIOSE/IHCEDIOS constructs a unit block for that data set in an area acquired by a GETMAIN, and places a pointer to it in the IHCUATBL entry. (This unit block, which is slightly different from ones created by IHCFIOSH/IHCEFIOS, is diagrammed in Figure 61.)

IHCDIOSE/IHCEDIOS next reads the Job File Control Block (JFCB) via a RDJFCB macro instruction. The appropriate fields in the JFCB are examined to determine if the user included a request for track overflow and a BUFNO subparameter in his DD statement for this data set. If he did, they are inserted into the DCB skeleton in the unit block. If BUFNO was not included or was other than 1 or 2, a value of 2 is inserted in the DCB skeleton. IHCDIOSE/IHCEDIOS next examines the data set disposition field of the JCFB. If the data set is new and the requested operation is a write, IHCDIOSE/IHCEDIOS must first format the data set before it can do the actual writing.

FORMATTING A NEW DATA SET:
IHCDIOSE/IHCEDIOS modifies the JFCB so that the disposition is old, and fills in the following fields in the DCB in the unit block:

DCB Field	Setting of field before OPEN
BUFNO	X'02' Two buffers
NCP	X'02' Two DECBs
DSORG	X'40' Set for DSORG=PS
MACR	X'0020' Normal BSAM WRITE
OPTCD	Set to X'00' or X'20' depending upon whether chained scheduling was not or was specified on the DD card as obtained from the JFCB.
DDNAME	Set to FTnnF001, where 'nn' is the DSRN.

Then an OPEN macro instruction, using BSAM, is issued (TYPE=J). The record length field, buffer address field, and DCB address field are filled in the DECB's. Then IHCDIOSE/IHCEDIOS issues sufficient WRITE macro instructions for fixed unblocked blank records to format the track(s). Record length and number specifications are taken from the DEFINE FILE parameter list pointed to by IHCUATBL.

The TRBAL field is used during BSAM writing to calculate whether there is enough room on the track for additional records after it has written the required number of fixed-length records. If the track is not full, data management does not create an R0 record and the OS utilities cannot process the data set. Therefore, if the track is not full, the library writes as many extra records as necessary until the track is complete.

The data set is then closed. The DCB is modified in the following way in order that it may be re-opened for BDAM and the actual writing.

DCB Field	New Setting for BDAM OPEN
NCP	X'00' Reset for BDAM
DSORG	X'02' DSORG=DA
MACR	X'29' BDAM update and check
MACR + 1	X'28' BDAM WRITE by ID
OPTCD	X'01' BDAM relative block address.

The procedure then is the same as opening an old data set (see below).

OPENING A DATA SET WHOSE DISPOSITION IS OLD: The data set is opened for BDAM, with the UPDAT option. In its open exit routine, IHCDIOSE/IHCEDIOS supplies default values (from the IHCUATBL entry) for those omitted by the user. After the open, IHCDIOSE/IHCEDIOS inserts into the DECB's the address(es) of the buffer(s) obtained during control block opening.

After doing this, or if the data set is already opened, IHCDIOSE/IHCEDIOS performs the following actions:

- Write: Upon initial branch, IHCDIOSE/IHCEDIOS does no writing at this time, but only fills the buffer with zeros and passes buffer address and buffer length back to IHCFCOMH/IHCECOMH so the latter may begin moving in the list items.

- Read: Upon initial branch, IHCDIOSE/IHCEDIOS gets the relative record number requested by the user, which has been passed along by IHCFCOMH/IHCECOMH. IHCDIOSE/IHCEDIOS examines the buffer to see if the record is already present. (This will be the case if the user previously requested a FIND for this record.) If not present, IHCDIOSE/IHCEDIOS issues a READ macro and, in either case issues a CHECK. After updating the associated variable in the parameter list to point to the record following the one just read, IHCDIOSE/IHCEDIOS returns to IHCFCOMH/IHCECOMH, passing the buffer address and length.

Successive Entries for List Items

WRITE OPERATION: When IHCFCOMH/IHCECOMH has filled the buffer with list items, it branches to IHCDIOSE/IHCEDIOS indicating a write request. IHCDIOSE/IHCEDIOS obtains the relative record number from the parameter list passed along by IHCFCOMH/IHCECOMH, and writes the record out via a WRITE macro instruction. It updates the associated variable in the parameter list to point to the record following the one just written. If single buffering is being used, it checks the write and returns to IHCFCOMH/IHCECOMH. If double buffering is being used, it postpones the check until its next call, and returns the address of the other buffer to IHCFCOMH/IHCECOMH.

READ OPERATION: IHCDIOSE/IHCEDIOS handles any further read requests from IHCFCOMH/IHCECOMH exactly as for the first (without checking for the data set being open).

Final Branch

WRITE OPERATION: IHCFCOMH/IHCECOMH calls IHCDIOSE/IHCEDIOS to write out the final buffer.

READ OPERATION: IHCFCOMH/IHCECOMH returns to the compiler-generated code without calling IHCDIOSE/IHCEDIOS.

Error Conditions

If IHCDIOSE/IHCEDIOS detects an input/output error condition, it performs in a manner similar to IHCFIOSH/IHCEFIOS by issuing a SYNADAF macro, using the resultant information to build a 218 error message, and passing control to IHCERRM.

IHCDIOSE/IHCEDIOS will also identify at one time or another the following error conditions:

231--the data set indicated by the caller is sequential rather than direct.
232--the record number requested is out of data set range.
233--the indicated record length exceeds 32K-1.
236--the read requested is for an uncreated data set.
237--the specified record length is incorrect.

In all these cases, IHCDIOSE/IHCEDIOS sets up the error message data and passes control to IHCERRM.

DIRECT ACCESS READ/WRITE WITH FORMAT

Requests for direct access reads and writes with format are handled by IHCFCOMH/IHCECOMH, with the assistance of IHCDIOSE/IHCEDIOS and IHCFCVTH. The actions of IHCDIOSE/IHCEDIOS are exactly the same as for unformatted direct access reads and writes. The actions of IHCFCOMH/IHCECOMH are exactly the same as for sequential read and write requests with format, except it calls IHCDIOSE/IHCEDIOS instead of IHCFIOSH/IHCEFIOS.

FIND

Implementation of the FIND statement is very similar to implementation of the opening branch for a direct access read (explained above). Control is passed from the compiler-generated code to IHCFCOMH/IHCECOMH and on to IHCDIOSE/IHCEDIOS. IHCDIOSE/IHCEDIOS opens the data set if need be, and then checks to see if the record is already in the buffer. If it is, IHCDIOSE/IHCEDIOS updates the associated variable. If not, it issues a READ macro. Then it returns through IHCFCOMH/IHCECOMH to the compiler-generated code. This READ begins filling the buffer. It is not checked until the next entry to IHCDIOSE/IHCEDIOS for this data set.

READ AND WRITE USING NAMELIST

Namelist reading and writing is handled by IHCNAMEL, with the assistance of IHCFIOSH/IHCEFIOS and IHCFCVTH. The compiler-generated object code branches only once to IHCNAMEL (to entry point FRDNL# for reads and to entry point FWRNL# for writes), passing the address of the namelist dictionary containing the user's specifications. IHCNAMEL uses this dictionary information to direct its operations, calling IHCFIOSH/IHCEFIOS to do the actual reading or writing, and the appropriate sections of IHCFCVTH to convert data and move it from buffer to user area or vice versa.

From the point of view of IHCFIOSH/IHCEFIOS and IHCFCVTH, a namelist read or write is no different than any other formatted sequential read/write operation. IHCNAMEL calls IHCFIOSH/IHCEFIOS once to initialize the data set and once to close it, and as many times in between to read or write as the namelist data requires.
IHCNAMEL calls IHCFCVTH as many times as the namelist data requires.

The namelist dictionary, which is the compiled version of the user's NAMELIST statement, consists of a 2-word namelist name field (right-justified and padded to the left with blanks), and as many entries as there were items in the NAMELIST definition. There are two types of entries: one for variables, and one for arrays. They are illustrated in Figures 40 and 41.

Read

IHCNAMEL first stores the END= and ERR= addresses, if they exist, in the proper locations in IHCFCOMH/IHCECOMH. This makes them available to IHCFIOSH/IHCEFIOS and IHCERRM if end-of-file or an input/output error occur.

IHCNAMEL searches through the data read by IHCFIOSH/IHCEFIOS looking for the namelist name that is located in the dictionary. When it locates the namelist name, it picks up the next data item. It now searches through the dictionary entries, looking for a matching variable or array name. When the name is located, IHCNAMEL obtains the associated specification information in that entry.

Processing of the constant in the input data now begins. Each initialization constant assigned to the variable or an array element is obtained from the input record. The appropriate conversion routine is selected according to the type of the variable or array element. Control is then passed to the conversion routine to convert the constant and to enter it into its associated variable or array element.

Note: One constant is required for a variable. A number of constants equal to the number of elements in the array is required for an array. A constant may be repeated for successive array elements if appropriately specified in the input record.

The process is repeated for the second and subsequent names in the input record. When an entire record has been processed, the next record is read and processed.

Processing is terminated upon recognition of the &END record. Control is then returned to the calling routine within the load module.

Write

IHCNAMEL takes the namelist name from the dictionary, puts it in the buffer, and has IHCFIOSH/IHCEFIOS write it out. The processing of the variables and arrays listed in the dictionary then begins.

The first variable or array name in the dictionary is moved to an output buffer followed by an equal sign. The appropriate conversion routine is selected according to the type of variable or array elements. Control is then passed to the conversion routine to convert the contents of the variable or the first array element and to enter it into the output buffer. A comma is inserted into the buffer following the converted quantity. If an array is being processed, the contents of its second and subsequent elements are converted, using the same conversion routine, and placed into the output buffer, separated by commas. When all of the array elements have been processed or if the item processed was a variable, the next name in the dictionary is obtained. The process is repeated for this and subsequent variable or array names.

If, at any time, the record length is exhausted, the current record is written and processing resumes in the normal fashion.

When the last variable or array has been processed, the contents of the current record are written, the characters &END are moved to the buffer and written, and control is returned to the calling routine within the load module.

Error Conditions

IHCNAMEL calls IHCERRM if it cannot find a name in the dictionary (error 222), if a name exceeds permissible length (221), if it cannot locate the required equal sign in the input data (223), or if a subscript is included for a variable or is out of range for an array (224).

STOP AND PAUSE (WRITE-TO-OPERATOR)

Stop

Control is passed by the compiler-generated code to the FSTOP section of IHCFCOMH/IHCECOMH. This section determines if there is a user message attached. If not, it simply branches to the IBEXIT section of IHCFCOMH/IHCECOMH to terminate load module execution. If there is a message, the FSTOP section issues the message to the console via SVC 35. It then branches to the IBEXIT section to terminate load module execution.

Pause

Control is passed by the compiler-generated code to the FPAUS section of IHCFCOMH/IHCECOMH. FPAUS issues a SVC 35 including the user's message or identifier, or "00000" if there was none. It then issues a WAIT to determine when the reply has been transmitted. After the operator or terminal user replies, IHCFCOMH/IHCECOMH returns control to the compiler-generated code.

BACKSPACE

Control is passed from the compiler-generated code to the FBKSP section of IHCFCOMH/IHCECOMH, which passes control to IHCFIOSH/IHCEFIOS.

For unblocked records, IHCFIOSH/IHCEFIOS issues a physical backspace (BSP) to position to the desired record. If 2 buffers are used, it must backspace twice to account for having read a record ahead. Before backspacing an output data set all WRITE requests are checked and an endfile mark is written by issuing a T-CLOSE. If the record form is V, it reads the record and examines the Segment Descriptor Word to determine if it has found the first segment. If it has, it issues another backspace. If it has not found the first segment, 2 backspaces are issued until the first segment is obtained, in which case it need only issue a final backspace.

For FB and VB records it must keep track of the location within the block of the record it wants. For the case of blocked records a BACKSPACE statement does not necessarily imply issuing a physical backspace request. A physical backspace is only required when the preceding logical record desired is in the block preceding the block presently in the buffer. IHCFIOSH/IHCEFIOS determines the length of the block read by subtracting the residual count in the CCW from the DCB blocksize. This information is used in calculating the proper logical record in the buffer to satisfy the FORTRAN BACKSPACE. Spanned records may require searching back through more than one physical record.

Control is returned to IHCFCOMH/IHCECOMH, which returns to the main program.

REWIND

The compiler-generated object code passes control to the FRWND section of IHCFCOMH/IHCECOMH, which passes control to IHCFIOSH/IHCEFIOS.

IHCFIOSH/IHCEFIOS issues a CLOSE macro with the REREAD option for the indicated data set. This has the effect of rewinding it. A FREEPOOL macro is issued to release the buffer space. Control returns through IHCFCOMH/IHCECOMH to the main program.

END-FILE

Control is passed by the compiler-generated object code to the FEOFM section of IHCFCOMH/IHCECOMH, which passes control to IHCFIOSH/IHCEFIOS.

If the previous operation for this data set was a read, IHCFIOSH/IHCEFIOS sets the DCBOFLGS bit to dummy a write operation. It issues a CLOSE macro with type T. This effects the writing of the end-of-file mark. (A 'T-CLOSE' rather than a full CLOSE is issued in order to handle any subsequent BACKSPACE requests.) A FREEPOOL macro is issued to release the buffer space. Return is through IHCFCOMH/IHCECOMH to the compiler-generated code.

ERROR HANDLING

The library is designed to handle the following error conditions:

- some compiler-detected source statement errors
- library-detected errors
- some program interrupts
- scheduled load module abnormal termination
- some user-defined and user-detected errors (only if extended error handling has been selected)

Library operations for interrupts and for errors it detects itself depend on whether the extended error handling facility was selected at program installation time.

The following library modules are concerned primarily with error handling:

- IHCADJST
- IHCERRM
- IHCFINTH/IHCEFNTH
- IHCFOPT
- IHCIBERH
- IHCSTAE
- IHCTRCH/IHCETRCH
- IHCUOPT

In addition, IHCFCOMH/IHCECOMH is used for initialization, loading, and termination;

IHCFCVTH is used for converting error message data; and IHCFIOSH/IHCEFIOS is used for printing error messages out.

COMPILER-DETECTED ERRORS: IHCIBERH

When the compiler examines and translates the user's source statements, it may recognize one to be faulty, and nonexecutable. At the corresponding location in the object code, the compiler inserts a branch to the library program IHCIBERH. The load module then executes in its usual fashion up to this point, when IHCIBERH gains control.

If the faulty statement has an Internal Statement Number (ISN), IHCIBERH translates it into hexadecimal and inserts it into itserror message--230. It also picks up the name of the user routine containing the faulty statement, and adds it to the message. After IHCERRM is utilized to have the message printed out, IHCIBERH goes to the IBEXIT section of IHCFCOMH/IHCECOMH to have load module execution terminated.

PROGRAM INTERRUPTS

Part of the library's initialization procedure is to issue a SPIE macro instruction, informing the system that the library wishes to gain control when certain program interrupts occur. The SPIE, issued by IHCFCOMH/IHCECOMH, specifies library control for the following interrupts:

6--specification*
9--fixed-point divide
11--decimal divide
12--exponent overflow
13--exponent underflow
15--floating-point divide

The exit routine address specified for all of the above is ARITH#, the beginning of IHCFINTH/IHCEFNTH. (If interrupts 2, 3, 4, 5, or 7 occur for the load module, the system begins abnormal termination processing. Codes 8, 10 and 14 are disabled when the task gains control, so these interrupts never occur.)

IHCFINTH/IHCEFNTH receives control from the system, which passes the address of the Program Interrupt Element (PIE) in register 1. IHCFINTH/IHCEFNTH first saves the interrupted program's registers 3-13 (the system saves only 14-2 in the PIE). IHCFINTH/IHCEFNTH next examines the old Program Status Word (PSW) in the PIE to see if the interrupt was precise or imprecise, and, if the latter, whether single or multiple. (Imprecise interrupts are explained more fully in the publication IBM System/360 Operating System: Supervisor and Data Management Services, Order No. GC28-6646.) This information is inserted in the error message--210. The specific interrupt type(s) is then determined.

Action for Interrupts 9, 11, 12, 13, and 15

IHCFINTH/IHCEFNTH sets the switch OVFIND or DVCIND in IHCFCOMH/IHCECOMH to indicate that one of the three divide checks or exponent overflow or underflow has occurred. (These switches are referenced by the routines IHCFOVER and IHCFDVCH.) When extended error handling is not in effect, IHCFINTH takes the following corrective actions:

9--nothing
11--nothing
15--if the operation is 0.0/0.0, the answer register(s) is set to 0.0; if the operation is X.Y/0.0 (X.Y≠0.0), the answer register(s) is set to the largest possible floating-point number
12--the result register(s) is set to the largest possible floating-point number
13--the result register(s) is set to 0.0; if the underflow resulted from an add or subtract operation, the condition code in the old PSW is set to 0.

Note that for corrective actions with 12, 13, and 15, it is necessary for IHCFINTH to first determine if the faulty instruction contains single or double precision operands.

IHCFCVTH is called (twice) to convert the error message contents, and IHCFIOSH is called to print it out. Then IHCFINTH returns to the system interrupt handler, and load module execution eventually resumes at the instruction following the one that caused the interrupt.

When extended error handling has been selected, IHCERRM is called to determine if the user desires his own corrective action for this error. (This procedure is described in the section "Extended Error Handling" below.) If no user action is specified, the standard actions described

---

*Issued only if the user selected the boundary alignment option at program installation time.

above are followed. In either instance, IHCERRM has the error message printed out.

Action for Interrupt 6

When a specification interrupt has occurred, IHCFINTH/IHCEFNTH loads IHCADJST, if not already loaded. After preparing the error message, it branches to IHCADJST passing the PIE and other information.

There is a great variety of error conditions that can cause a specification interrupt. (They are explained in the publication IBM System/360: Principles of Operation, Order No. GA22-6821.) IHCADJST is designed to correct only one--the misalignment of operand data in core. For any other condition, IHCADJST causes an abnormal termination by cancelling the SPIE, backing up the PSW pointer to the instruction that caused the original interrupt,* and returning to the system.

When IHCADJST determines that it has a data boundary alignment problem to correct, it calls IHCFINTH/IHCEFNTH to have the error message (210) written out. Next IHCADJST issues a new SPIE, for protection (4) and addressing (5) exceptions, so that if an interrupt occurs while it is trying to fetch a copy of the operand data, its own special section--PAEXCPT--will gain control. If one of these exceptions does occur, PAEXCPT calls IHCFINTH/IHCEFNTH to have the error message written, and then causes abnormal termination as described above.

After IHCADJST has properly aligned the data in a temporary storage location and is ready to try to re-execute the original instruction, it issues yet another SPIE (overlaying the previous) for interrupts 4, 7, 9, 11, 12, 13, and 15. If re-execution of the original instruction is successful, and the R1 field of the instruction re-executed was 14, 15, 0, or 1, IHCADJST puts the new contents of that register into the PIE. If the condition code was changed by the re-execution, the new condition code is put into the PSW located in the PIE. If the instruction re-executed was a ST, STE, or STD, the data is moved to the correct location in the load module. The original load module SPIE is re-established, and control is returned directly to the supervisor, rather than via IHCFINTH/IHCEFNTH. Note that the correction of data misalignment is only temporary; the permanent locations of user variables remain the same.

If re-execution of the original instruction causes a second interrupt, control is given to EXCPTN in IHCADJST. For code 4 and 7, IHCFINTH/IHCEFNTH is called to have the error message written, and IHCADJST then causes abnormal termination in the manner described above. For the other exceptions, the original PIE is reconstructed, the original SPIE re-established, and control passed back to IHCFINTH/IHCEFNTH to process this new interrupt in its usual fashion.

---

*In the case of instruction misalignment, when it is determined the next instruction is also misaligned and will cause abnormal termination just as well, the PSW pointer is not changed.

LIBRARY-DETECTED ERRORS

A number of the library routines examine their operational data for flaws. For example, most of the mathematical routines check to see if the arguments are within specified ranges; IHCFCVTH, in some cases, sees whether the data it is asked to convert is actually in the form specified.

When a library routine finds an error, it sets up a branch to IHCERRM. If extended error handling has been selected for the library, this is a separate module. If not, it is simply the entry point name for module IHCTRCH (and module IHCERRM does not exist). Without extended error handling, library-detected errors are almost always treated as terminal conditions.

Without Extended Error Handling

IHCTRCH is passed the number of the error condition and the message if one is to be printed for this particular case.* IHCTRCH's functions are to have the error message printed and, more significantly, to create the traceback map and have it printed. IHCTRCH employs IHCFCVTH to convert information to printable decimal and hexadecimal format, and IHCFIOSH to do the actual printing. Then IHCTRCH calls the IBEXIT section of the IHCFCOMH to terminate load module execution. Condition 218 is an exception if the user has specified an ERR= parameter on his READ source statement. In this case, IHCTRCH picks up this address from IHCFCOMH and passes control to it.

---

*Errors 211-214, 217, 219, 220, and 231-237 have only IHCxxxI printed out, without any text.

The traceback information printed consists of routine names in the load module internal calling sequence, the ISN of each branch instruction, and each routine's registers 14-1. In most cases, the map begins with the routine that called the library module that detected the error, then lists the routine that called that caller, and so on back to the compiler-generated main program. In the case of the mathematical routines, however, the traceback map begins with that mathematical routine detecting the error. IHCTRCH gets the map information by using register 13 as a starting point and working its way back through the linked save areas. Because some library routines (e.g., IHCFCOMH) do not use standard saving procedures, the tracing can become rather complicated.

IHCTRCH terminates the trace when it finds it has done one of three things:

1. reached the compiler-generated main routine
2. reached 13 levels of call
3. found a calling loop

In the second and third cases, it prints 'TRACEBACK TERMINATED', and in all cases prints the main program entry point.

IHCTRCH goes immediately to the IBEXIT section of IHCFCOMH for termination if it is entered a second time. This can happen if an input/output error occurs while IHCFIOSH is trying to print IHCTRCH's output.

With Extended Error Handling

When a library routine detects an error and extended error handling is available, it branches to the error monitor routine IHCERRM. The operation of this routine is explained below in the section "Extended Error Handling Facility."

ABNORMAL TERMINATION PROCESSING

When the load module has been scheduled by the system for abnormal termination, the library attempts to have any output buffer contents written out.

During load module initialization, IHCFCOMH/IHCECOMH issues a STAE macro, specifying that if the load module is ever scheduled for abnormal termination, the address EXITRTN1 in IHCFCOMH/IHCECOMH should be given control by the system.

When EXITRTN1 does gain control, it loads IHCSTAE from the link library and branches to it, passing along the system input/output status codes it received. These are:

Code (in Register 6)	Meaning
0	Active input/output was quiesced and is restorable
4	Active input/output was halted and is not restorable
8	No active input/output at abnormal termination time
12	No space available for work area

IHCSTAE looks at this code and determines which action it will take.

Codes 4 and 12

After using IHCFCVTH to convert the abnormal termination code (either system or user), the location of the STAE control block and the load module PSW into hexadecimal, IHCSTAE inserts them into its error messages (240), and issues the messages via WTO macro instructions. Then it returns to the supervisor, indicating (with a 0 in register 15) the abnormal termination is to be completed.

Codes 0 and 8

After using IHCFCVTH to convert the abnormal termination code (either system or user), the location of the STAE control block and the load module PSW into hexadecimal, IHCSTAE inserts them into its messages. Then, IHCSTAE returns to the supervisor, indicating with a 4 in register 15 that a retry attempt (RETRY in IHCSTAE) is wanted. When this section gains control, it first issues another STAE macro instruction specifying a new exit routine, so that in the event of a new abnormal termination condition arising, looping will not occur. Next, the system's STAE work area is tested to see whether there is active restorable input/output or no input/output active at all. If the former, SVC 17 is issued (RESTORE macro) to prepare for the resumption of the load module's input/output activity.

In both cases, IHCERRM is called to print message 240 and a traceback map. Before calling IHCERRM, however, IHCSTAE searches through the chained save areas (beginning with the supervisor's) to determine whether or not the abnormal termination condition will prevent the traceback map from listing the routine causing the abnormal termination; if it will, IHCSTAE appends a statement to this effect in its error message.

If extended error handling is not in effect, IHCTRCH (entry point IHCERRM) exits to the IBEXIT section of IHCFCOMH/IHCECOMH. If extended error handling is in effect, IHCERRM returns to IHCSTAE, which calls the IBEXIT section of IHCFCOMH/IHCECOMH. The IBEXIT section calls IHCFIOSH/IHCEFIOS to complete pending output requests--that is, flush the buffers. (This is the normal load module termination process.) IHCFCOMH/IHCECOMH finally returns to the supervisor.

In the event of a second abnormal termination condition occurring, control is given to EXITRTN3 in IHCSTAE. No retry is attempted. Messages are issued via WTO macro instructions, and control is returned to the supervisor to complete abnormal termination.

EXTENDED ERROR HANDLING FACILITY

Three routines are centrally involved with extended error handling operations. They are:

1. IHCUOPT--the option table
2. IHCFOPT--the routine available to the user to reference and modify the option table
3. IHCERRM--the routine that handles the errors according to the option table entries

In addition, IHCETRCH is used to produce traceback maps. (When extended error handling has not been selected, IHCFOPT does not exist at all, IHCERRM does not exist as a module but only as an entry point in IHCTRCH, and IHCUOPT is only 8 bytes long.)

Option Table--IHCUOPT

The format of the option table is illustrated in Figures 62 through 64. The table is referenced by displacement. It is sequential, but begins (after a preface) with error 207--the lowest library error. There is an entry for every number from 207 to 301, although the library recognizes no error condition for some of them -- e.g. 239 (they are reserved for future use). Thus, the entry for error 258 is (258-207+1)x8 bytes into the table (allowing for the preface). A few library error numbers (900-904) are not in the table.

Certain values are inserted in the option table at system generation time. These original values are listed in Figure 65. The user has the power to alter some of these values temporarily--that is, alter the copy in main storage for the duration of the load module--by using FORTRAN source statements. All the library error entries except 230 and 240 can be altered.

Altering the Option Table--IHCFOPT

The user's source statement requests for referencing and altering the option table are handled by IHCFOPT, which is branched to directly by the compiler-generated code. IHCFOPT has three entry points for its three functions: ERRSAV, ERRSTR, and ERRSET.

ERRSAV AND ERRSTR: These two functions are quite simple. They are passed an error number and an address. ERRSAV takes a copy of the requested error number entry from the table and places it at the indicated address. ERRSTR takes the new 8-byte entry from the indicated user address and inserts it in the table, overlaying the original entry.

ERRSAV and ERRSTR both first check to see that the error number is within the table range. If it is not, they issue message 902, employing IHCFCVTH and IHCEFIOS in the process. ERRSTR also checks bit 1 of byte 4 of the old table entry to make sure modification is permissible. If it is not, it issues message 903, with the help of IHCFCVTH and IHCEFIOS. Return is to the calling program in all cases.

ERRSET: ERRSET also modifies table entries, but is more flexible than ERRSTR. It is passed either five or six parameters, and takes the following actions:

- The error number: a reference only.
- A new limit count for entry field one: contents are moved in as is, unless the count is greater than 255, in which case the field is set to 0, or unless

the count is 0, in which case no action is taken.

- A new message count for entry field two: contents are moved in as is, unless they are negative or zero. If they are negative, the field is set to 0; if they are 0, no action is taken.

- Traceback requested or suppressed: if 1, bit 6 of entry field four is turned off; if 0, it is turned on; if any other number, no action is taken.

- A user exit routine address, or absence thereof, for entry field five: the value is moved in as is.

- (Optional parameter) - Either an error number higher than one in the first parameter, or, if the first parameter is error 212, a request for print control: in the first case, all entries from the lower number to the higher are altered as indicated; in the second case, if a 1, bit 0 of field four is set to 1, if not a 1, it is set to 0.

ERRSET checks to make sure that the error number entry or entries indicated are within the table range. If not, it issues message 902, using IHCFCVTH and IHCEFIOS. ERRSET also checks to make sure that the entry or entries permit modification. If they do not, it issues message 903 using IHCFCVTH and IHCEFIOS.

Error Monitor--IHCERRM

The error monitor is called in the following three cases:

1. When a library module has discovered an error condition during its processing (entry point IHCERRM)

2. When the user's program has detected one of the user-defined errors (302-899) and wishes to handle it according to his option table entry (entry point ERRMON)

3. During normal load module termination processing, to give the error count summary (entry point IHCERRE)

In the first two cases, the error monitor consults the corresponding entry in the option table IHCUOPT to determine what actions it will take for this particular error condition.

After using the error number passed to it to locate the corresponding option table entry, the error monitor updates the error count field and compares it to the limit field. If the limit is now exceeded, it begins the termination process. This involves having IHCEFIOS print out message 900 and the error message passed by the caller (if the option table indicates it is desired), and having IHCETRCH produce the traceback map (if the option table so indicates). Finally, the IBEXIT section of IHCECOMH is given control. (The error monitor may be entered again to give the error summary. See "Error Summary.")

If the error count limit is not yet exceeded, the error monitor has the caller error message and the traceback map produced (if the table so indicates), using IHCEFIOS and IHCETRCH, respectively. Then it sees whether or not a user exit routine is specified. If it is, IHCERRM branches to it passing along data supplied by the routine that detected the error. The nature of this data depends on the error detected.

The user routine is required to return to the error monitor, indicating that it has either performed corrective action itself (a 1 in the first parameter), or wants standard library corrective action (a 0 in the first parameter). The error monitor issues a message reporting on this status, and then returns to its original caller, passing the correction code. The caller either resumes its normal processing, or does its standard correction before continuing.

If the error monitor finds no user exit address, it returns to the caller requesting standard correction.

SPECIAL CONDITIONS: The error monitor will not allow recursive usage. If it is entered a second time before its current processing is finished, it issues message 901 and begins the termination procedure. The error monitor also checks to make sure the error number specified is within the option table range; if it is not, it issues message 902.

The error monitor performs an additional step when it finds the error to be 218. In this case, after going to the user exit routine if there was one, IHCERRM determines from IHCECOMH if the user has specified an ERR= address on his READ source statement. If so, IHCERRM branches to it.

For error 218, the error monitor issues a FREEMAIN macro instruction to free the message area the calling routine acquired.

ERROR SUMMARY: The summary routine (entry IHCERRE) simply loops through the option table, finding those entries for which errors have occurred during load module execution, and putting the error numbers and their accumulated counts in the message. It uses IHCFCVTH for conversion and IHCEFIOS for printing. If IHCEFIOS has identified an error condition for the object error unit, the summary is skipped.

Extended Error Handling Trackback--IHCETRCH

IHCETRCH performs in the same manner as IHCTRCH, with these three exceptions:

1. IHCETRCH is called by IHCERRM, rather than directly by the error-detecting routine.
2. IHCETRCH does not have the error-detecting routine's message printed out, since this is done by IHCERRM.
3. IHCETRCH can also be called by the user, through a source statement calling its entry point ERRTRA. A traceback requested in this way is not necessarily connected with any error condition. IHCETRCH returns to the user program.

CONVERSION

Routine IHCFCVTH, the library conversion routine, is called by IHCFCOMH/IHCECOMH to convert user input/output data under FORMAT control, by IHCNAMEL to convert user input/output data under NAMELIST control, and by service routines (such as IHCFDUMP) and error handling routines (such as IHCERRM and IHCTRCH) to convert output data into printable (EBCDIC) hexadecimal and/or decimal form.

IHCFCVTH is divided into a number of subroutines (see Table 49). Each subroutine is designed to convert a particular type of input or output data. The library routine calling IHCFCVTH selects which conversion operation it wants, and branches to the appropriate subroutine. The calling routine passes the address of the existing data item, the address at which to place the result, the length, scale factor, and decimal point location of the existing data item, and other related information.

The subroutine then converts and moves the data item, and returns to its caller.

MATHEMATICAL AND SERVICE ROUTINES

The library contains a large number of mathematical routines, and some service routines. When a particular routine has been requested by the user in his source program (by entry point name), or when the compiler has recognized an implicit need for a mathematical function, it is branched to directly from the compiler-generated code.

Table 49. IHCFCVTH Subroutine Directory

Subroutine	Function
FCVAI	Reads alphameric data.
FCVAO	Writes alphameric data.
FCVCI	Reads complex data.
FCVCO	Writes complex data.
FCVDI	Reads double precision data with an external exponent.
FCVDO	Writes double precision data with an external exponent.
FCVEI	Reads real data with an external exponent.
FCVEO	Writes real data with an external exponent.
FCVFI	Reads real data without an external exponent.
FCVFO	Writes real data without an external exponent.
FCVGI	Reads general type data.
FCVGO	Writes general type data.
FCVII	Reads integer data.
FCVIO	Writes integer data.
FCVLI	Reads logical data.
FCVLO	Writes logical data.
FCVZI	Reads hexadecimal data.
FCVZO	Writes hexadecimal data.

MATHEMATICAL ROUTINES

The mathematical routines are generally independent of the other library programs (except when they detect errors or cause arithmetic-type type program exceptions). they perform their calculations, possibly with the assistance of another mathematical routine or two, and return directly to the compiler-generated code. The internal logic of these routines is documented in the publication IBM System/360 Operating System: FORTRAN IV Library--Mathematical and Service Subprograms, Order No. GC28-6818, under the section "Algorithms."

SERVICE SUBROUTINES

IHCFDVCH (Entry Name DVCHK)

The function of IHCFDVCH is to test the status of the divide check indicator switch (DVCIND--located in IHCFCOMH/IHCECOMH) and return an answer in the location specified in the call. This switch is turned on (set to X'FF' by the library's interrupt handler) when it finds a divide exception has occurred. IHCFDVCH inserts a 1 in the calling program's answer location if the switch is on, or a 2 if it is off.* The answer location is the argument variable inthe original FORTRAN statement CALL DVCHK(arg). Its address is pointed to by Register 1 when IHCFDVCH gains control.

If the DVCIND switch is on, IHCFDVCH turns it off (set to X'00'; if off, it is left off. IHCFDVCH returns to the calling program.

IHCFOVER (Entry Name OVERFL)

IHCFOVER tests for overflow and underflow, and performs in a manner similar to IHCFDVCH. The switch it tests is OVFIND--which is also found in IHCFCOMH/IHCECOMH, and set by the library interrupt handler. OVFIND set to X'FF' indicates overflow has occurred, X'01' indicates underflow, X'00' indicates neither. IHCFOVER sets the caller's answer location to 1 for overflow, 3 for underflow, and 2 for neither.

If on, OVFIND is turned off; if off. left off. IHCFOVER returns to the calling program.

IHCFSLIT (Entry Names SLITE, SLITET)

IHCFSLIT performs two functions: sets the pseudo-sense lights (entry SLITE), and reports back to the caller on their status (entry SLITET).

The four pseudo-sense lights are four bytes in IHCFSLIT labelled SLITES. These switches are not connected with any system switches, nor directly with any system condition. They are internal to the load module, and have meaning only to the FORTRAN user, who, employing IHCFSLIT, both sets and interprets them.

SETTING THE SWITCHES: SLITE either turns off all the switches (sets them to X'00'), or turns on one (sets it to X'FF'). When the argument passed to it is 0, SLITE turns all switches off. When the argument is 1-4, it turns on the corresponding switch--that is, an argument of 2 turns on the second (from left) byte of SLITES.

TESTING THE SWITCHES: SLITET is passed two parameters, the first indicating the particular switch to be tested, and the second pointing to a location for its answer. SLITET returns the answer 1 if it finds the switch on, and 2 if it is off. If it finds the switch on, it turns it off; if it is off, it is left off.

ERROR CONDITIONS: Both SLITE and SLITET first test their arguments for correct range. For SLITE, this must be 0-4; for SLITET, 1-4. When an argument is in error, they get the address of the integer output section of IHCFCVTH (FCVIO) from IHCFCOMH/IHCECOMH, and branch to it to have the error message contents converted. Then IHCFSLIT branches to IHCERRM (see the section on library-detected errors).

If extended error handling is not in effect, IHCERRM goes to the IBEXIT section of IHCFCOMH/IHCECOMH to terminate load module execution. If extended error handling is in effect, and IHCFSLIT, upon regaining control, finds the user did no special fixup, IHCFSLIT's standard corrective action is as follows:

SLITE: no action at all
SLITET: answer returned to caller is 2; no switches are changed

---

*Before checking the switch, both IHCFDVCH and IHCFOVER issue the special no-operation BCR 15,0, which drains pipe-line models (e.g., Models 91 and 195) to ensure sequential execution.

IHCFEXIT (Entry Name EXIT)

IHCFEXIT simply branches to the IBEXIT section of IHCFCOMH/IHCECOMH, which then terminates load module execution in its usual way.

IHCFDUMP (Entry Names DUMP and PDUMP)

IHCFDUMP's function is to have printed out on the object error unit the storage contents specified in the call, in the format specified. The absolute storage location of each request is also printed out.

The call parameters are in this form:

```
DC      AL4(A1)
DC      AL4(B1)
DC      AL4(F1)
    .
    .
    .
DC      AL4(An)
DC      AL4(Bn)
DC      XL1'FF',AL3(Fn)
```

where A and B are addresses of the outer limits of the storage to be dumped, and F is either the integer format number itself, or the address of a location containing the number. The specifications are:

```
0 = hexadecimal
1 = LOGICAL*1
2 = LOGICAL*4
3 = INTEGER*2
4 = INTEGER*4
5 = REAL*4
6 = REAL*8
7 = COMPLEX*8
8 = COMPLEX*16
9 = literal
```

If the user passes any other number, IHCFDUMP chooses 0 (hexadecimal) as a default format.

The procedure is indentical for DUMP and PDUMP, except for two things:

- if DUMP finds an input/output corrective action routine is in process, it functions normally; PDUMP, nowever, instead of processing, goes to section ERR904 in IHCFCOMH/IHCECOMH to print error message 904 and to terminate load module execution. (An input/output corrective action routine in process is indicated by the first byte of SAVE in IHCFCOMH/IHCECOMH set to anything other than X'FF'.)
- after normal processing, DUMP goes to the IBEXIT section of IHCFCOMH/IHCECOMH to terminate load module execution; PDUMP, however, returns to the caller for continued execution.

IHCFDUMP uses IHCFCVTH and IHCFIOSH/IHCEFIOS to assist in its operations. After getting the address of IHCFIOSH/IHCEFIOS from IHCFCOMH/IHCECOMH, IHCFDUMP branches to initialize for printing. It next moves a section to be dumped into the IHCFIOSH/IHCEFIOS buffer, and determines the format type requested.* It passes this information to the FCVZO part of IHCFCVTH ('Z' output), for conversion. Lastly, it branches to IHCFIOSH/IHCEFIOS to print out the line. IHCFDUMP loops in this manner until it exhausts the calling list.

If, during the printing, IHCFIOSH/IHCEFIOS indicates it has encountered an input/output error, IHCFDUMP skips the remainder of its work.

TERMINATION

Every compiler-generated program ends with a branch to the FSTOP section of IHCFCOMH/IHCECOMH (see Table 51 for IHCFCOMH/IHCECOMH branches and subroutines). This section is a termination procedure that:

- puts the return code passed it into register 15.
- if extended error handling has been specified, calls IHCERRM to have the error summary produced.
- calls IHCFIOSH/IHCEFIOS to close sequential files (IHCFIOSH/IHCEFIOS in turn calls IHCDIOSE/IHCEDIOS to close any direct access files).
- deletes IHCADJST, if it has been loaded.
- cancels the SPIE, restoring the old PICA if there was one.
- either

  a. cancels the STAE and returns to the supervisor if IHCSTAE has not been loaded (i.e., no abnormal termination has been scheduled)
  b. cancels the STAE and issues an ABEND macro instruction if entry is from IHCSTAE

The above termination procedure is used both for the normal end of load module

---

*IHCFDUMP expects the format type requested to correspond to the format of the data in main storage. Therefore, asking it to print out an INTEGER variable in REAL format, for example, will result in a garbled dump.

execution and for most instances of library-initiated premature termination. The only exceptions occur in IHCSTAE, when control is sometimes returned directly to the supervisor, bypassing the above procedure.

Unit number (DSRN) being used for current operation				n[1] x 16 bytes
ERRMSG DSRN[2]	READ DSRN[3]	PRINT DSRN[4]	PUNCH DSRN[5]	4 bytes
UBLOCK01 field[6]				4 bytes
DSRN01 default values[7]				8 bytes
LIST01 field[8]				4 bytes
.				.
.				.
.				.
UBLOCKn field[6]				4 bytes
DSRNn default values[7]				8 bytes
LISTn field[8]				4 bytes

[1]n is the maximum number of units that can be referred to by the FORTRAN LOAD MODULE. The size of the unit table is equal to (8 + n x 16) bytes.

[2]Unit number (DSRN) of error output device.

[3]Unit number (DSRN) of input device for a read of the form: READ b,list.

[4]Unit number (DSRN) of output device for a print operation of the form: PRINT b,list.

[5]Unit number (DSRN) of output device for a punch operation of the form: PUNCH b,list.

[6]The UBLOCK field contains either a pointer to the unit block constructed for unit number n if the unit is being used at object time, or a value of 1 if the unit is not being used.

[7]This field contains DCB default values, which are inserted into the DCB if the user does not supply them. They are detailed in Figure 59. Only IHCFIOSH/IHCEFIOS gets its default values from this field.

[8]If the unit is defined as a direct access data set, the LIST field contains a pointer to the parameter list that defines the direct access data set. Otherwise, this field contains a value of 1.

Figure 58. IHCUATBL: The Data Set Assignment Table

Table 50. DCB Default Values

	Sequential Data Sets					Direct Access Data Sets		
ddname	RECFM[1]	LRECL[2]	BLKSIZE	DEN	BUFNO	RECFM	LRECL or BLKSIZE	BUFNO
FT03Fxxx	U	--	800	2	2	FA	The value specified as the maximum size of a record in the DEFINE FILE statement.	2
FT05Fxxx	F	80	80	--	2	F		2
FT06Fxxx	UA	132	133	--	2	F		2
FT07Fxxx	F	80	80	--	2	F		2
all others	U	--	800	2	2	F		2

[1]For records not under FORMAT control, the default is VS.
[2]For record not under FORMAT control, the default is 4 less than shown.

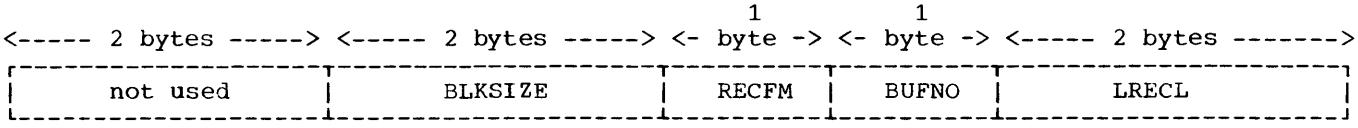


Figure 59. DSRN Default Value Field of IHCUATBL Entry

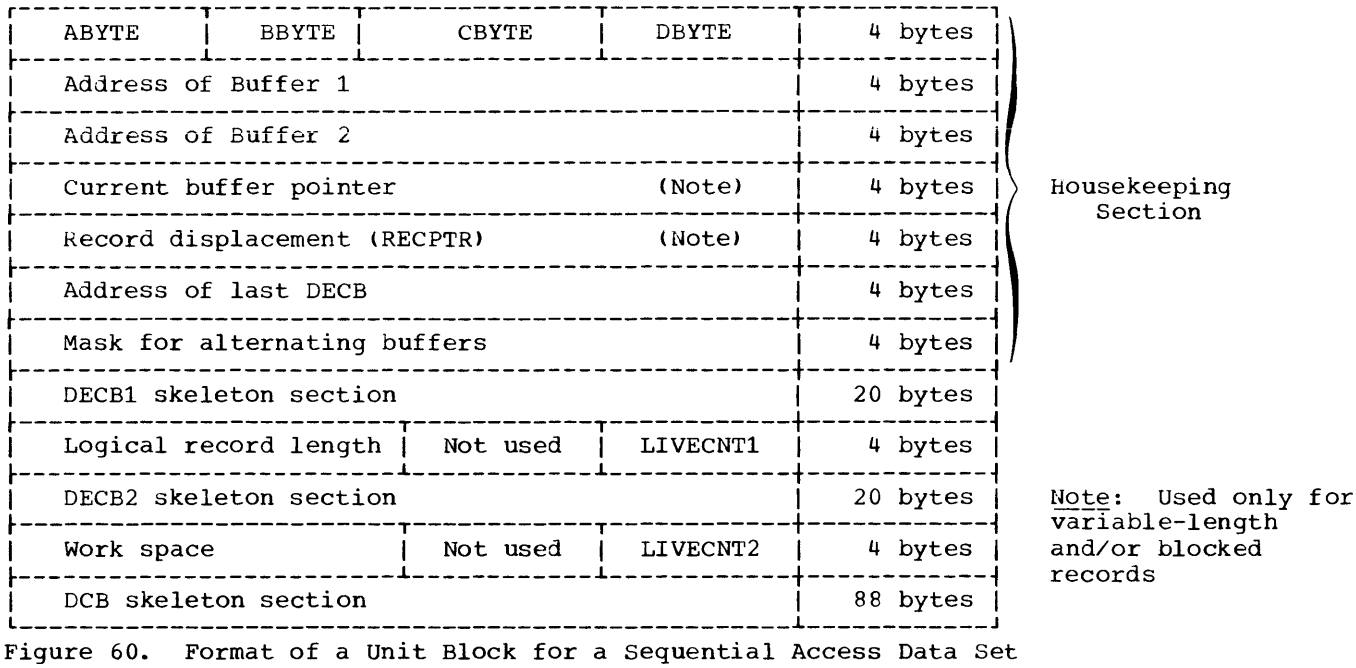


Figure 60. Format of a Unit Block for a Sequential Access Data Set

- ABYTE. This field, containing the data set type passed to subprogram IHCFIOSH/IHCEFIOS by IHCFCOMH/IHCECOMH, is set to one of the following:

Setting	Meaning
F0	Input data set which is to be processed under format control.
FF	Output data set which is to be processed under format control.
00	Input data set which is to be processed without format control.
0F	Output data set which is to be processed without format control.

- BBYTE. This field contains bits that are set and examined by IHCFIOSH/IHCEFIOS during its processing. The bits and their meanings, when on, are as follows:

Bit	Meaning
0	exit to subroutine IHCFCOMH/IHCECOMH on input/output error
1	input/output error occurred
2	current buffer indicator
3	not used
4	end-of-current buffer indicator
5	blocked data set indicator
6	variable record format switch
7	not used

- CBYTE. This field also contains bits that are set and examined by subroutine IHCFIOSH/IHCEFIOS. The bits and their meanings, when on, are as follows:

Bit	Meaning
0	data control block opened
1	data control block not T-closed
2	data control block not previously opened
3	buffer pool attached
4	data set not previously rewound
5	not used
6	concatenation occurring; reissue READ
7	data set is DUMMY

- DBYTE. This field contains bits that are set and examined by IHCFIOSH/IHCEFIOS during the processing of an input/output operation involving a backspace request. The bits and their meanings, when on, are as follows:

Bit	Meaning
0	a physical backspace has occurred
1	previous operation was BACKSPACE
2	not used
3	end-of-file routine should retain buffers
4-5	not used
6	END FILE followed by BACKSPACE
7	not used

- Address of Buffer 1 and Address of Buffer 2. These fields contain pointers to the two input/output buffers obtained during the opening of the data control block for this data set.

- Current Buffer Pointer. This field contains a pointer to the input/output buffer currently being used.

- Record Offset (RECPTR). This field contains a pointer to the current logical record within the current buffer.

- Address of Last DECB. This field contains a pointer to the DECB last used.

- Mask for Alternating Buffers. This field contains the bits which enable an exclusive OR operation to alternate the current buffer pointer.

- DECB Skeleton Sections (DECB1 and DECB2): The DECB (data event control block) skeleton sections are blocks of main storage within the unit block. They have the same format as the DECB constructed by the control program for an L format of an S-type READ or WRITE macro instruction (see the publication IBM System/360 Operating System: Supervisor and Data Management Macro Instructions, Order No. GC28-6647). The various fields of the DECB skeleton are filled in by subprogram IHCFIOSH;

the completed block is referred to when IHCFIOSH issues a read/write request to BSAM. The read/write field is filled in when the OPEN macro is being executed.

- Logical Record Length: This is the LRECL of the current data set. It is inserted by IHCFIOSH/IHCEFIOS during its open exit routine.

- LIVECNT1 and LIVECNT2. These fields indicate whether any input/output operation performed for the data set is unchecked. (A value of 1 indicates that a previous read or write has not been checked; a value of 0 indicates that the previous read or write operation on that DECB has been checked.)

- Work Space. This field is used to align the logical record length of a variable record segment on a fullword boundary.

- DCB Skeleton Section: The fields of this skeleton for DCB are filled in partly by IHCFIOSH/IHCEFIOS, and partly by the system as a result of an OPEN macro instruction by IHCFIOSH/IHCEFIOS.

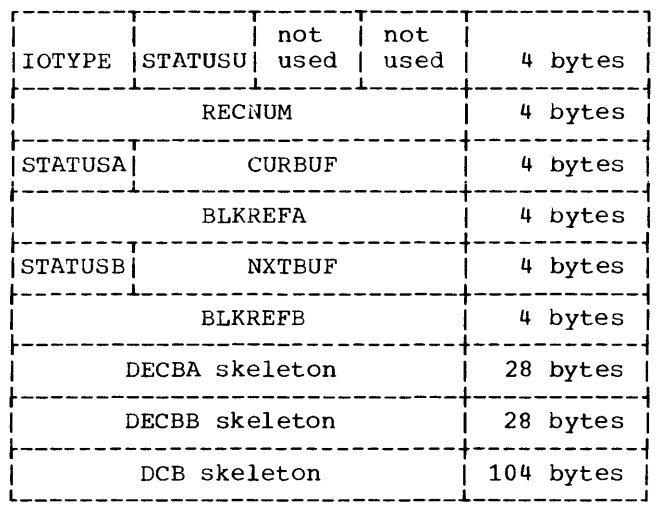


Figure 61. Format of a Unit Block for a Direct Access Data Set

The meanings of the various unit block fields are outlined below.

- IOTYPE: This field, containing the data set type passed to subprogram IHCDIOSE by the IHCFCOMH subprogram, can be set to one of the following:

Setting	Meaning
F0	input data set requiring a format
FF	output data set requiring a format
00	input data set requiring a format
0F	output data set not requiring a format

- STATUSU: This field specifies the status of the associated unit number. The bits and their meaning when on are:

Bit	Meaning
0	data control block for data set is open for BSAM
1	error occurred
2	two buffers are being used
3	data control block for data set is open for BDAM
4-5	10 - U format specified in DEFINE FILE statement 01 - E format specified in DEFINE FILE statement 11 - L format specified in DEFINE FILE statement
6-7	not used

Note: Subprogram IHCDIOSE refers only to bits 1, 2, and 3.

- RECNUM: This field contains the number of records in the data set as specified in the parameter list for the data set in a DEFINE FILE statement. It is filled in by the file initialization section after the data control block for the data set is opened.

- STATUSA: This field specifies the status of the buffer currently being used. The bits and their meanings when on are:

Bit	Meaning
0	READ macro instruction has been issued
1	WRITE macro instruction has been issued
2	CHECK macro instruction has been issued
3-7	not used

- CURBUF: This field contains the address of the DECB skeleton currently being used. It is initialized to contain the address of the DECBA

skeleton by the file initialization section of IHCDIOSE after the data control block for the data set is opened.

- BLKREFA: This field contains an integer that indicates either the relative position within the data set of the record to be read, or the relative position within the data set at which the record is to be written. It is filled in by either the read or write section of subprogram IHCDIOSE prior to any reading or writing. In addition, the address of this field is inserted into the DECBA skeleton by the file initialization section of IHCDIOSE after the data control block for the data set is opened.

- STATUSB: This field specifies the status of the next buffer to be used if two buffers are obtained for this data set during data control block opening. The bits and their meanings are the same as described for the STATUSA field. However, if only one buffer is obtained during data control block opening, this field is not used.

- NXTBUF: This field contains the address of the DECB skeleton to be used next if two buffers are obtained during data control block opening. It is initialized to contain the address of the DECBB skeleton by the file initialization section of subprogram IHCDIOSE after the data control block for the data set is opened. However, if only one buffer is obtained during data control block opening, this field is not used.

- BLKREFB: The contents of this field are the same as described for the BLKREFA field. It is filled in either by the read or the write section of subprogram IHCDIOSE prior to any reading or writing. In addition, the address of this field is inserted into the DECBB skeleton by the file initialization section of IHCDIOSE after the data control block for the data set is opened. However, if only one buffer is obtained during data control block opening, this field is not used.

- DECBA Skeleton: This field contains the DECB (data event control block) skeleton to be used when reading into or writing from the current buffer. It is the same form as the DECB constructed by the control program for an L form of an S-type READ or WRITE macro instruction under BDAM (see the publication IBM System/360 Operating System: Supervisor and Data Management Macro Instructions, Order No. GC28-6647).

The various fields of the DECBA skeleton are filled in by the file initialization section of subprogram IHCDIOSE after the data control block for the data set is opened. The completed DECB is referred to when IHCDIOSE issues a read or a write request to BDAM. For each input/output operation, IHCDIOSE supplies IHCFCOMH with the address of and the size of the buffer to be used for the operation.

- DECBB Skeleton: The DECBB skeleton is used when reading into or writing from the next buffer. Its contents are the same as described for the DECBA skeleton. The DECBB skeleton is completed in the same manner as described for the DECBA skeleton. However, if only one buffer is obtained during data control block opening, this field is not used.

- DCB Skeleton: This field contains the DCB (data control block) skeleton for the associated data set. It is of the same format as the DCB constructed by the control program for a DCB macro instruction under BDAM (see the publication IBM System/360 Operating System: Supervisor and Data Management Macro Instructions Order No. GC28-6447).

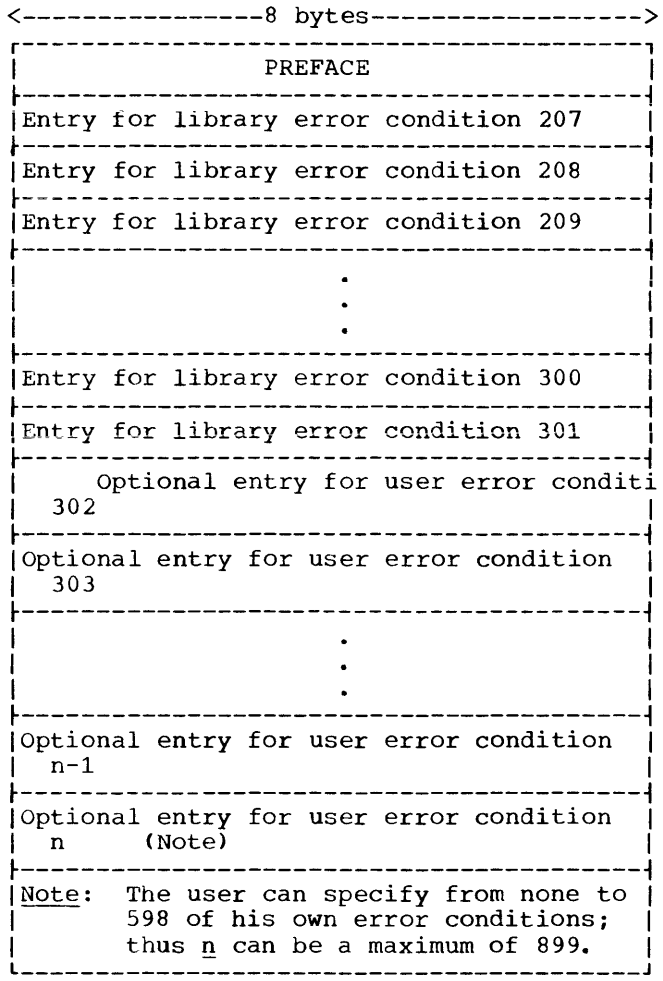
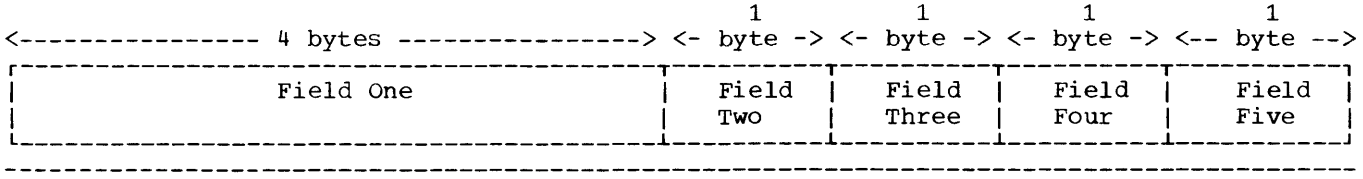


Figure 62. General Form of the Option Table (IHCUOPT)



Field Contents

One: The number of entries in the option table. This is 95 plus the total number of user-supplied error conditions.

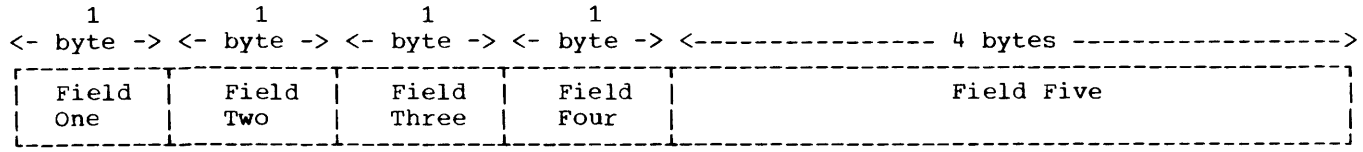
Two: Bit one indicates whether boundary alignment was selected. 1=yes; 0=no. (Bits 0 and 2-7 are reserved for future use.)

Three: Indicates whether extended error handling was selected. X'FF'=no; X'00'=yes.

Four: Contains a decimal 10. This is the number of times the boundary alignment error message will be printed when extended error handling has not been specified.

Five: Reserved for future use.

Figure 63. Preface of the Option Table (IHCUOPT)



Field	Contents
One:	The number of times the library should allow this error to occur before terminating load module execution. A value of zero means unlimited occurrence. (Trying to set the field to greater than 255 results in its being set to zero.)
Two:	The number of times the corresponding error message is to be printed before message printing is suppressed. A value of zero means no message is to be printed.
Three:	The number of times this error has already occurred in execution of the present load module.
Four:	Bit Meaning 0 If control character is supplied for overflow lines, set to 1. If control character is not supplied for overflow lines, set to 0. 1 If this table entry can be user-modified, set to 1. If this table entry cannot be user-modified, set to 0. 2 If more than 255 errors of this type have occurred so that 255 should be added to Field Three, set to 1. If less than 255 errors of this type have occurred, set to 0. 3 If buffer contents are to be printed with error messages, set to 1. If buffer contents are not to be printed, set to 0. 4 Reserved for future use. 5 If error message is to be printed for every occurrence, set to 1. If error message is not to be printed, set to 0. 6 If traceback map is to be printed, set to 1. If traceback map is not to be printed, set to 0. 7 Reserved for future use.
Five:	The address of the user's exit routine. If one is not supplied (in other words, if library is to take its own standard corrections), the final bit is set to 1.

Figure 64. Composition of an Option Table Entry

```
    1         1         1         1
<- byte -> <- byte -> <- byte -> <- byte -> <---------------- 4 bytes ------------------>
┌---------┬---------┬---------┬---------┬-------------------------------------------┐
|  Field  |  Field  |  Field  |  Field  |                Field Five                 |
|  One    |  Two    |  Three  |  Four   |                                           |
└---------┴---------┴---------┴---------┴-------------------------------------------┘
```

Field Contents

One: Set to 10, except for errors 208, 210 and 215, which are set to 0 (unlimited), and for errors 217 and 230, which are set to 1.

Two: Set to 5, except for error 210, which is set to 10, and for errors 217 and 230, which are set to 1.

Three: Set to 0.

Four:

Bit	Setting
0	0
1	1, except for errors 230 and 240
2	0
3	0
4	0
5	0
6	1
7	0

Five: Set to 1.

Note: These system generation values are also inserted initially into any user error entries.

Figure 65. Original Values of Option Table Entries

Table 51. IHCFCOMH/IHCECOMH Transfer and Subroutine Table

Displacement from IBCOM#	Branches to Section	Function of Routine
0	FRDWF	Opening section, formatted READ
4	FWRWF	Opening section, formatted WRITE
8	FIOLF	I/O list section, formatted list variable
12	FIOAF	I/O list section, formatted list array
16	FENDF	Closing section, formatted READ or WRITE
20	FRDNF	Opening section, nonformatted READ
24	FWRNF	Opening section, nonformatted WRITE
28	FIOLN	I/O list section, nonformatted list variable
32	FIOAN	I/O list section, nonformatted list array
36	FENDN	Closing section, nonformatted READ or WRITE
40	FBKSP	Implements the BACKSPACE source statement
44	FRWND	Implements the REWIND source statement
48	FEOFM	Implements the ENDFILE source statement
52	FSTOP	Write-To-Operator, terminate job
56	FPAUS	Write-To-Operator, resume execution
64	IBFINT	Load module initialization
68	IBEXIT	Load module termination

Chart 23. IHCFCOMH/IHCECOMH (Part 1 of 4)

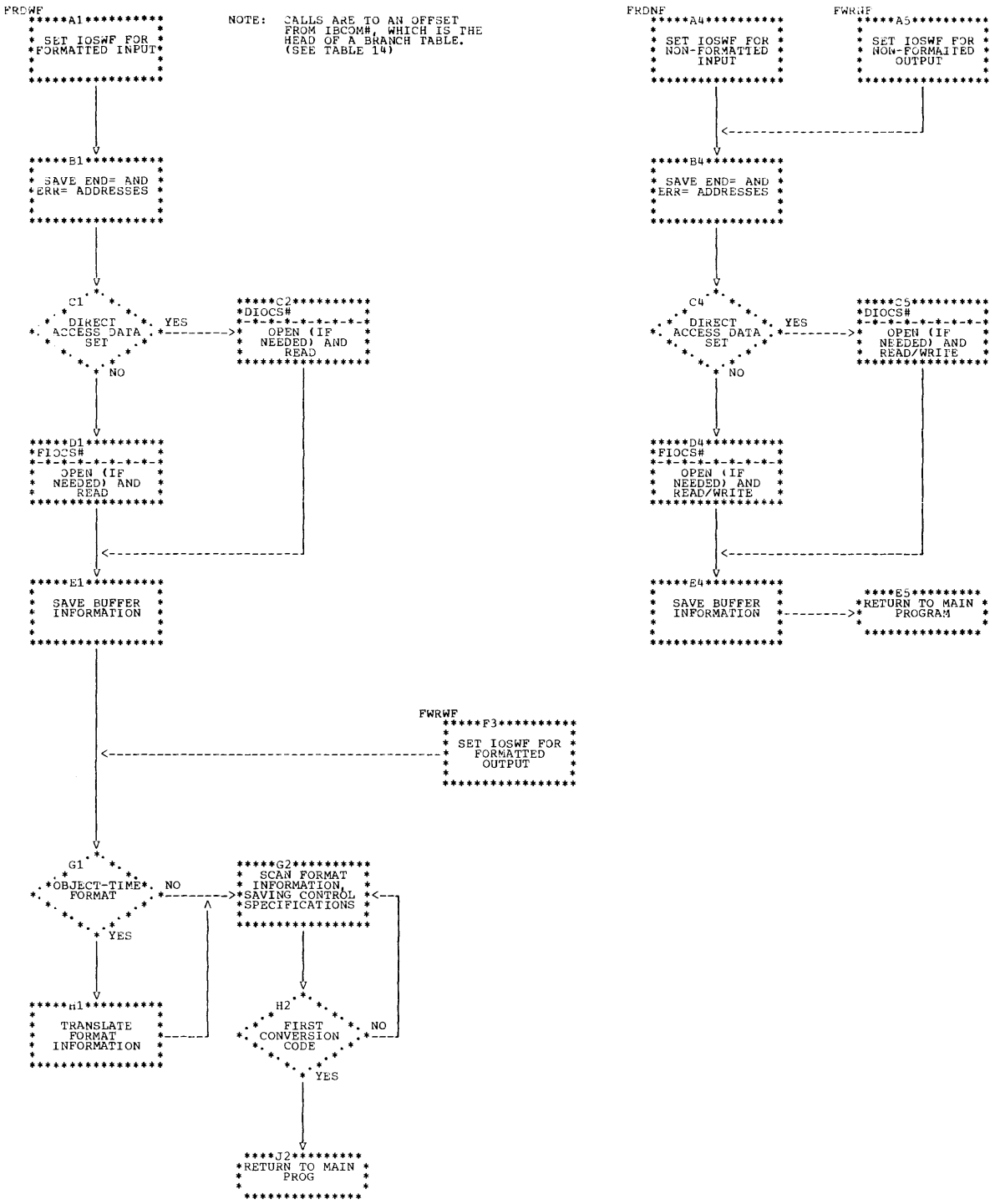


Chart 23. IHCFCOMH/IHCECOMH (Part 2 of 4)

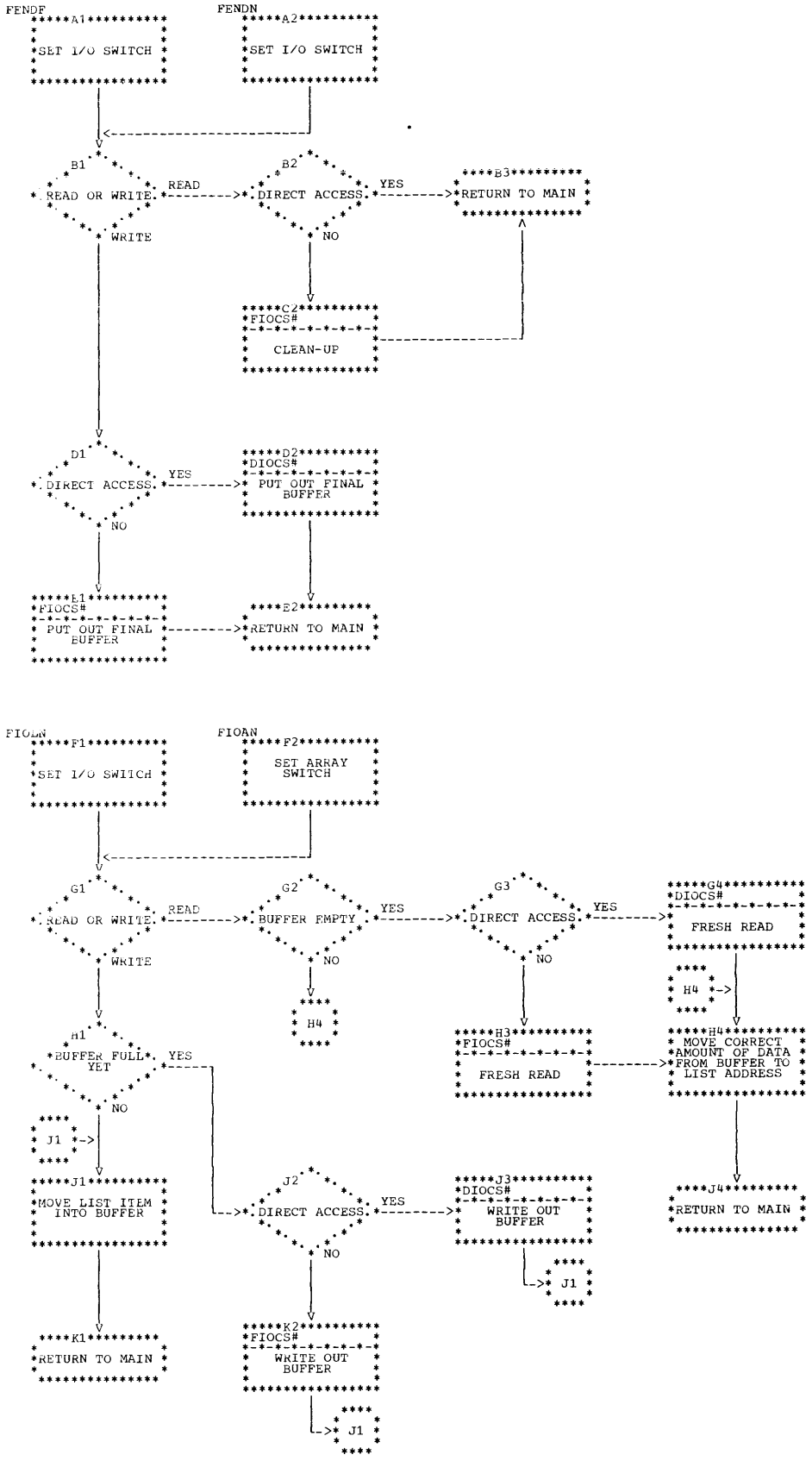
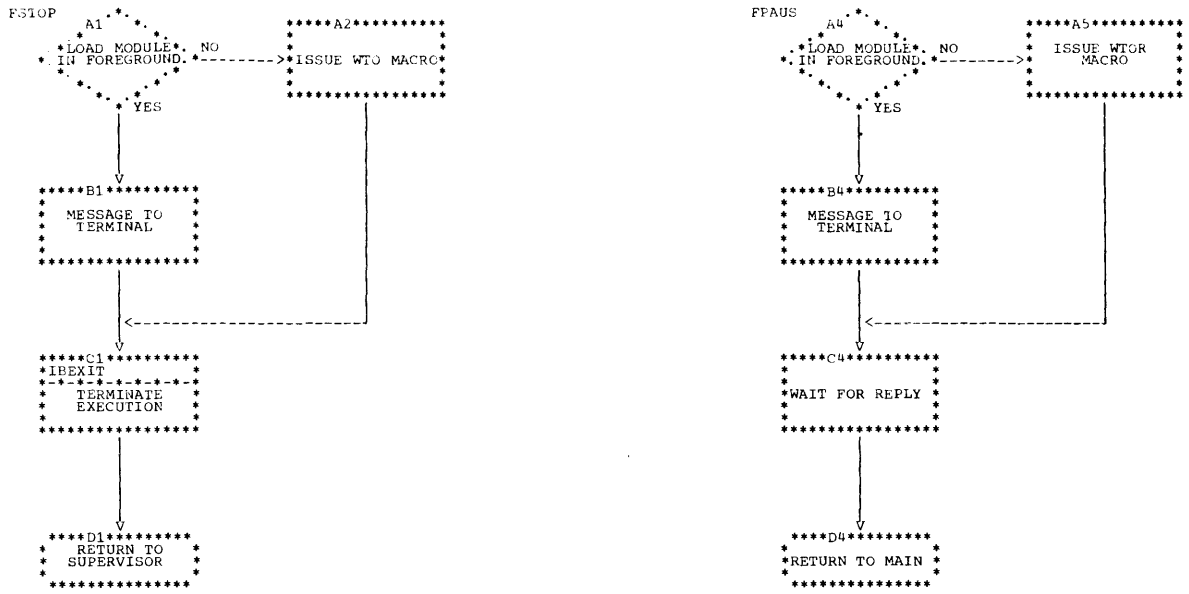


Chart 23. IHCFCOMH/IHCECOMH (Part 3 of 4)



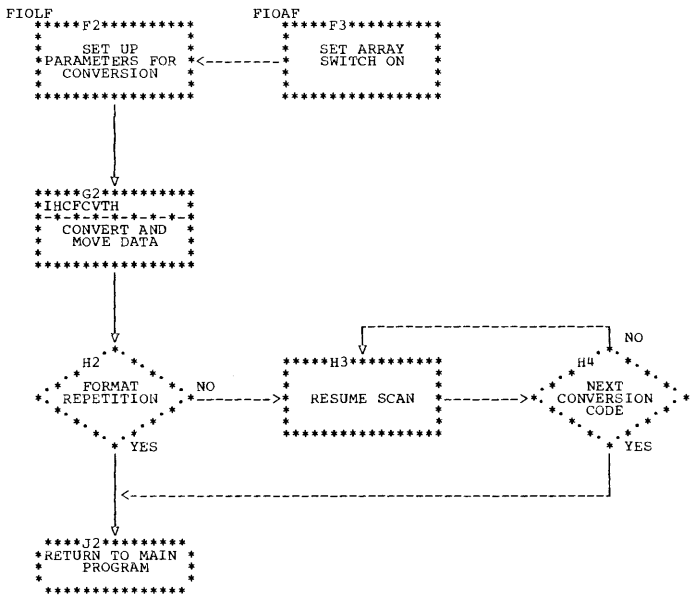


Chart 23. IHCFCOMH/IHCECOMH (Part 4 of 4)

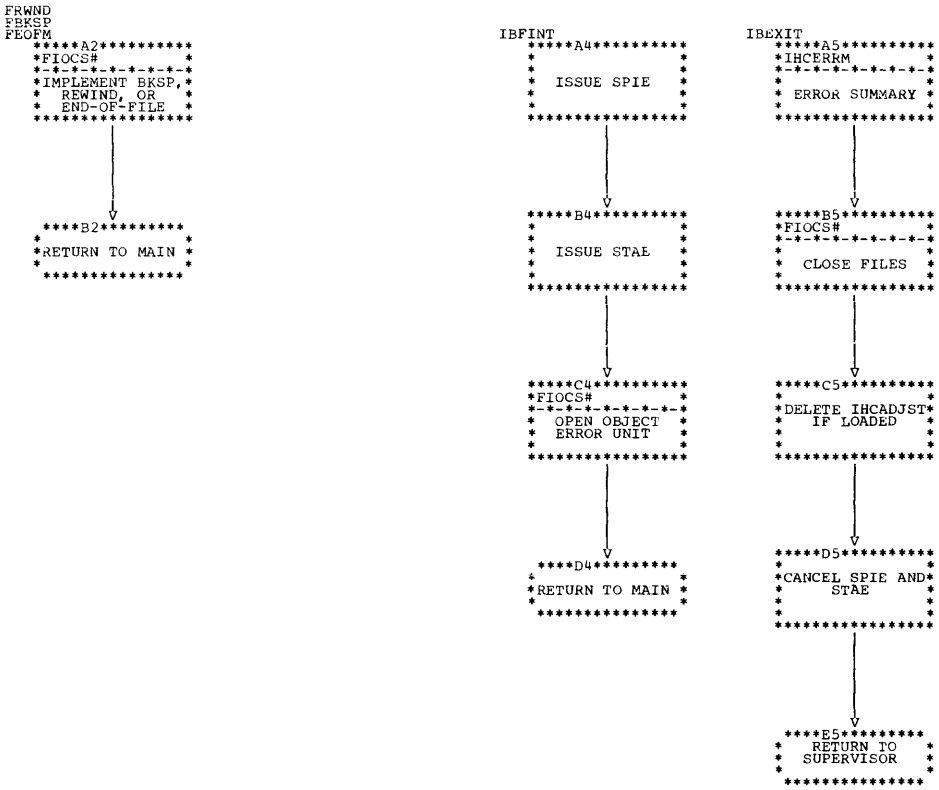


Chart 24. IHCFIOSH/IHCEFIOS (Part 1 of 2)

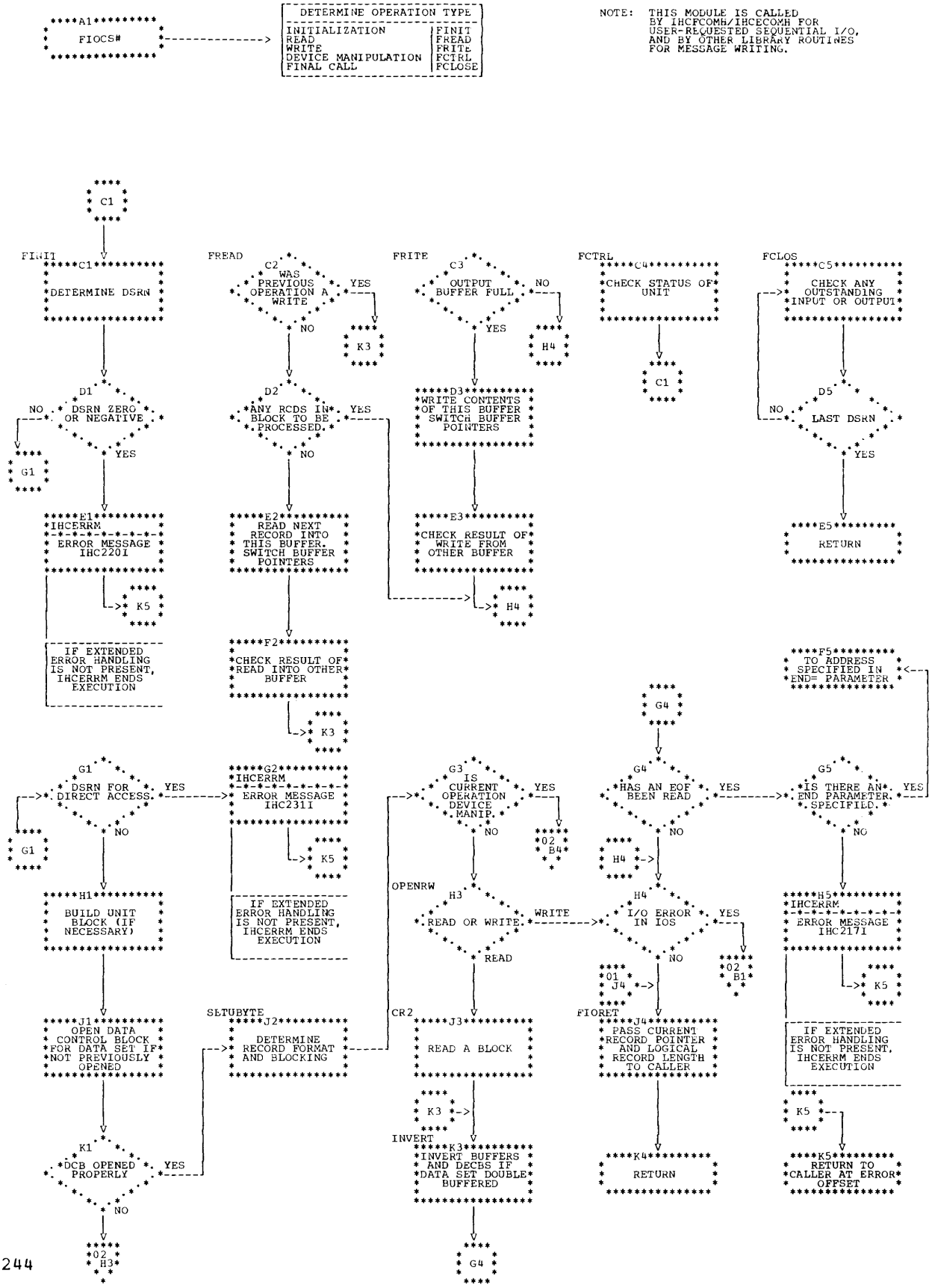


Chart 24. IHCFIOSH/IHCEFIOS (Part 2 of 2)

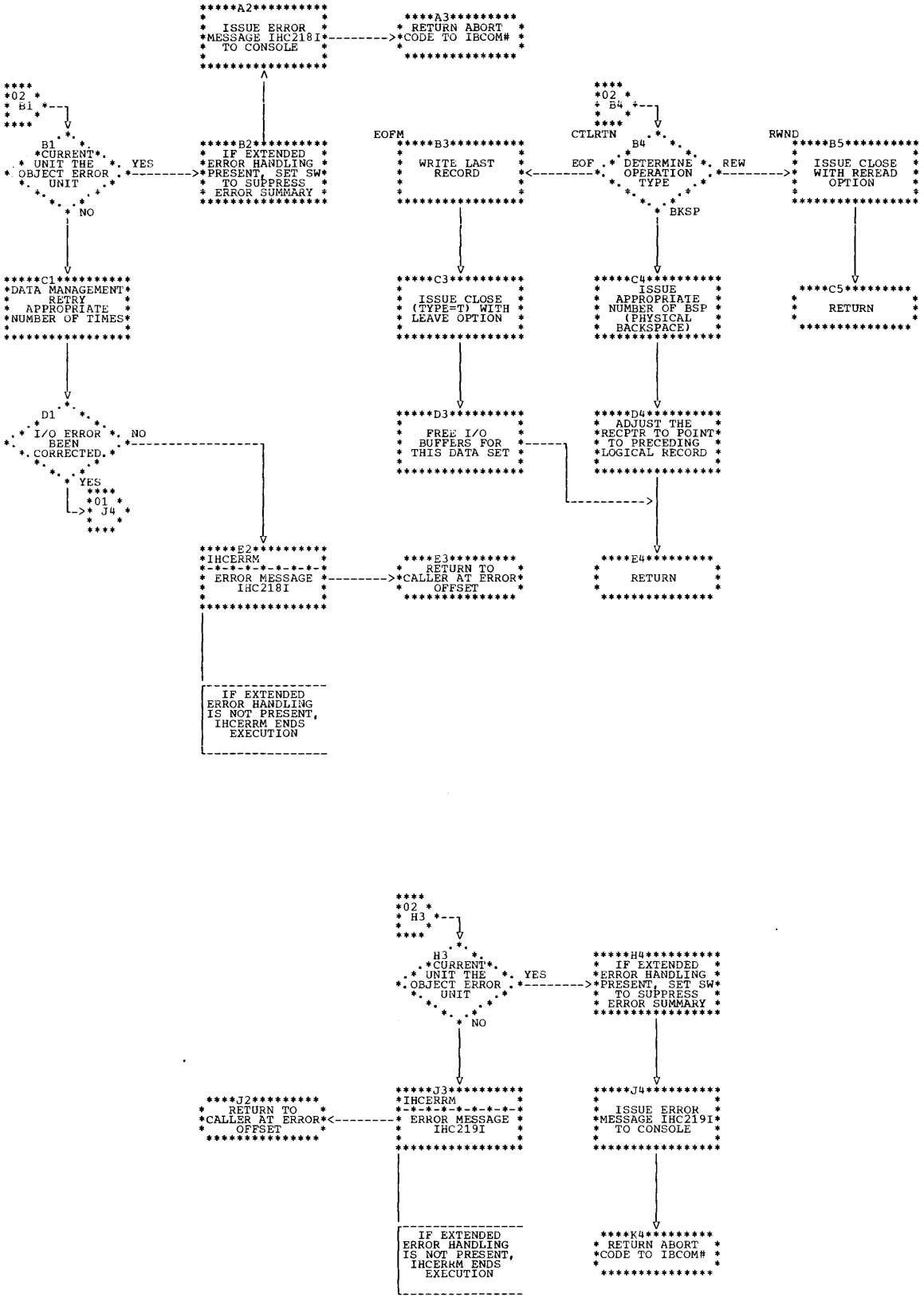


Chart 25. IHCDIOSE/IHCEDIOS (Part 1 of 5)

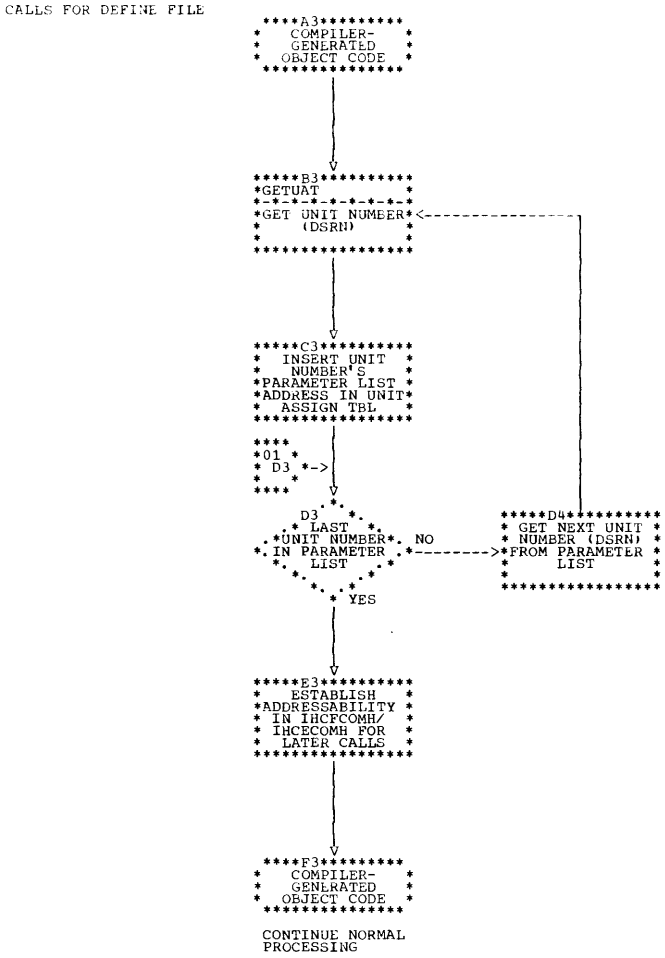


Chart 25. IHCDIOSE/IHCEDIOS (Part 2 of 5)

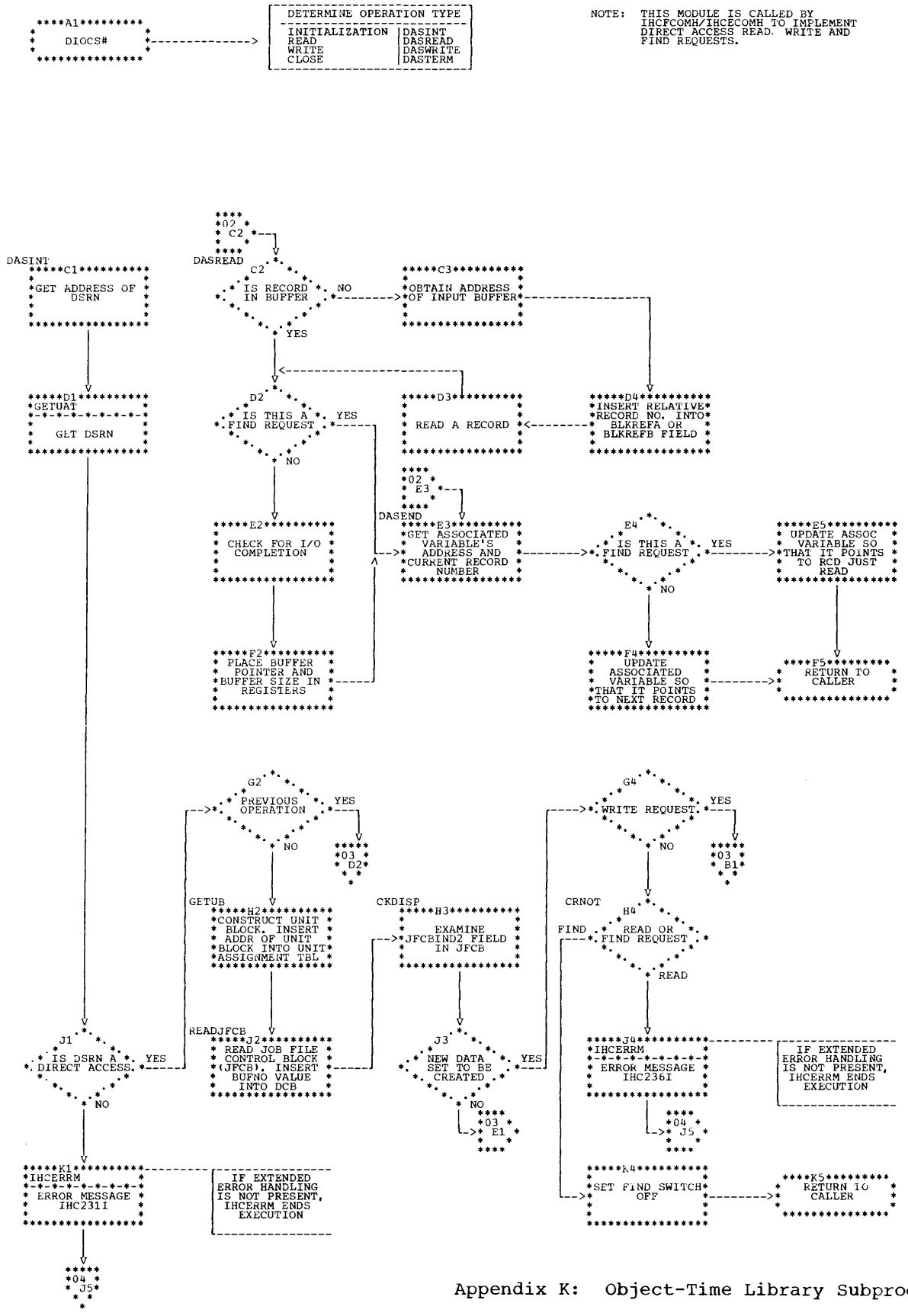


Chart 25. IHCDIOSE/IHCEDIOS (Part 3 of 5)

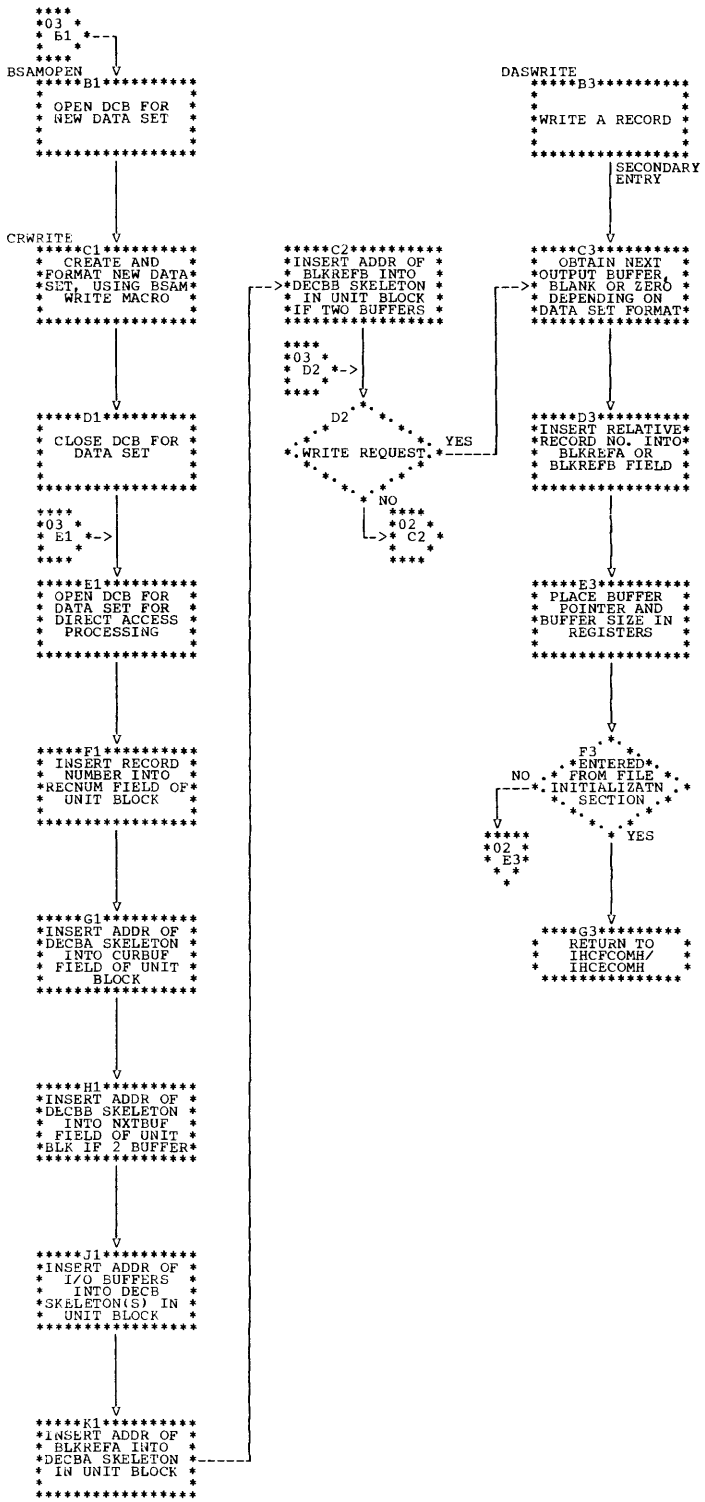


Chart 25. IHCDIOSE/IHCEDIOS (Part 4 of 5)

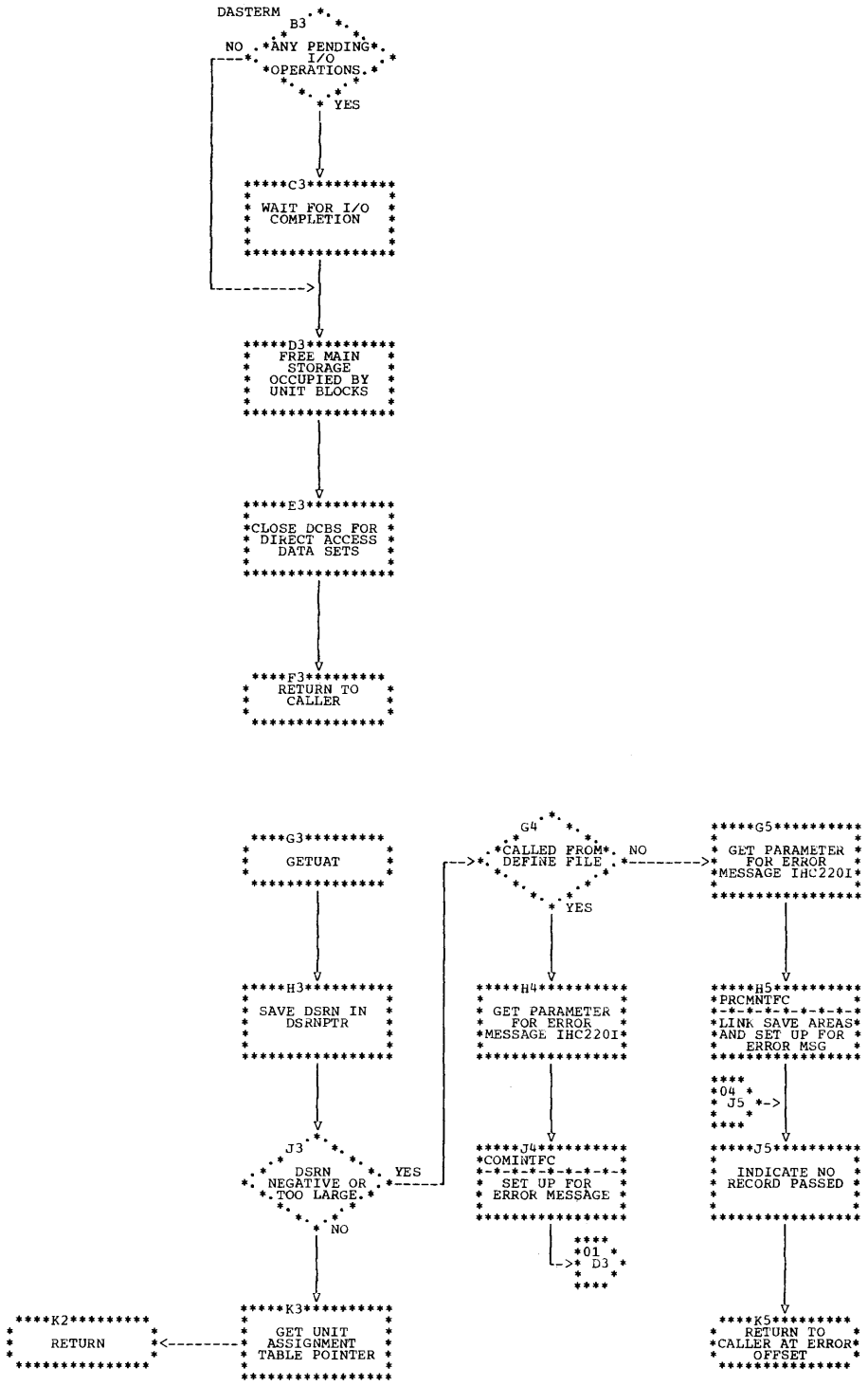


Chart 25. IHCDIOSE/IHCEDIOS (Part 5 of 5)

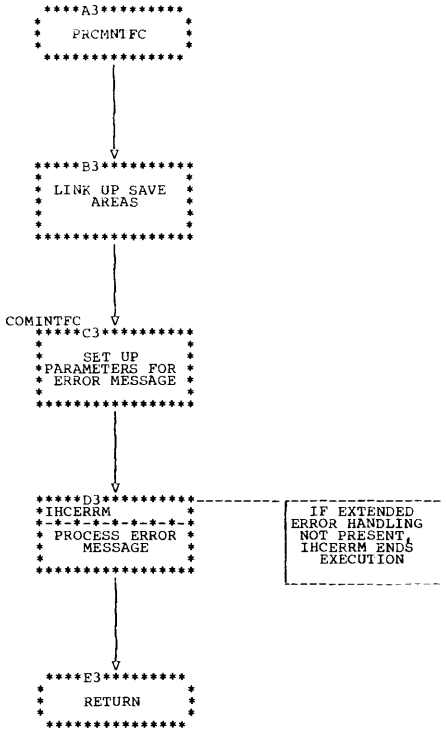
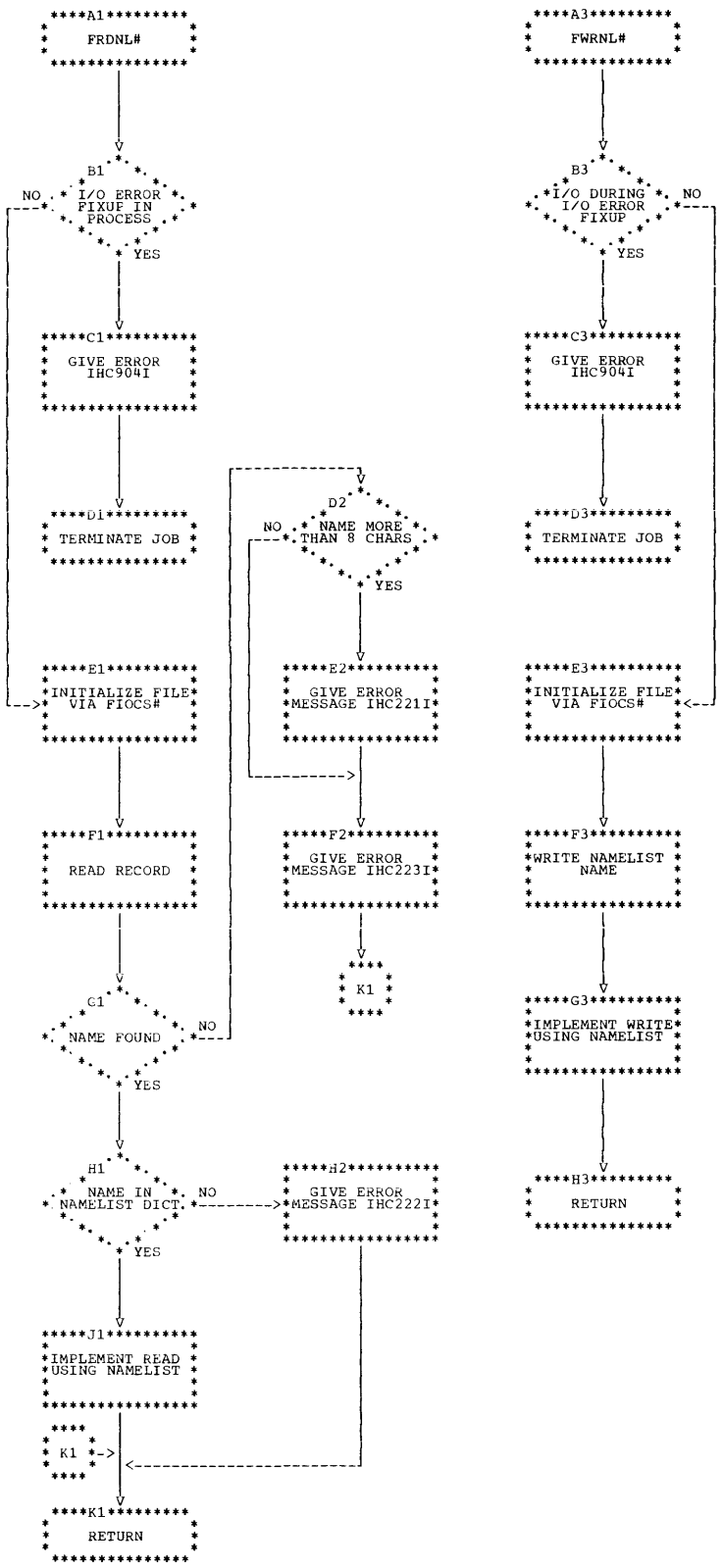


Chart 26. IHCNAMEL



NOTE: THIS MODULE IS CALLED BY THE COMPILER-GENERATED CODE TO IMPLEMENT NAMELIST I/O REQUESTS.

Chart 27. IHCFINTH/IHCEFNTH (Part 1 of 3)

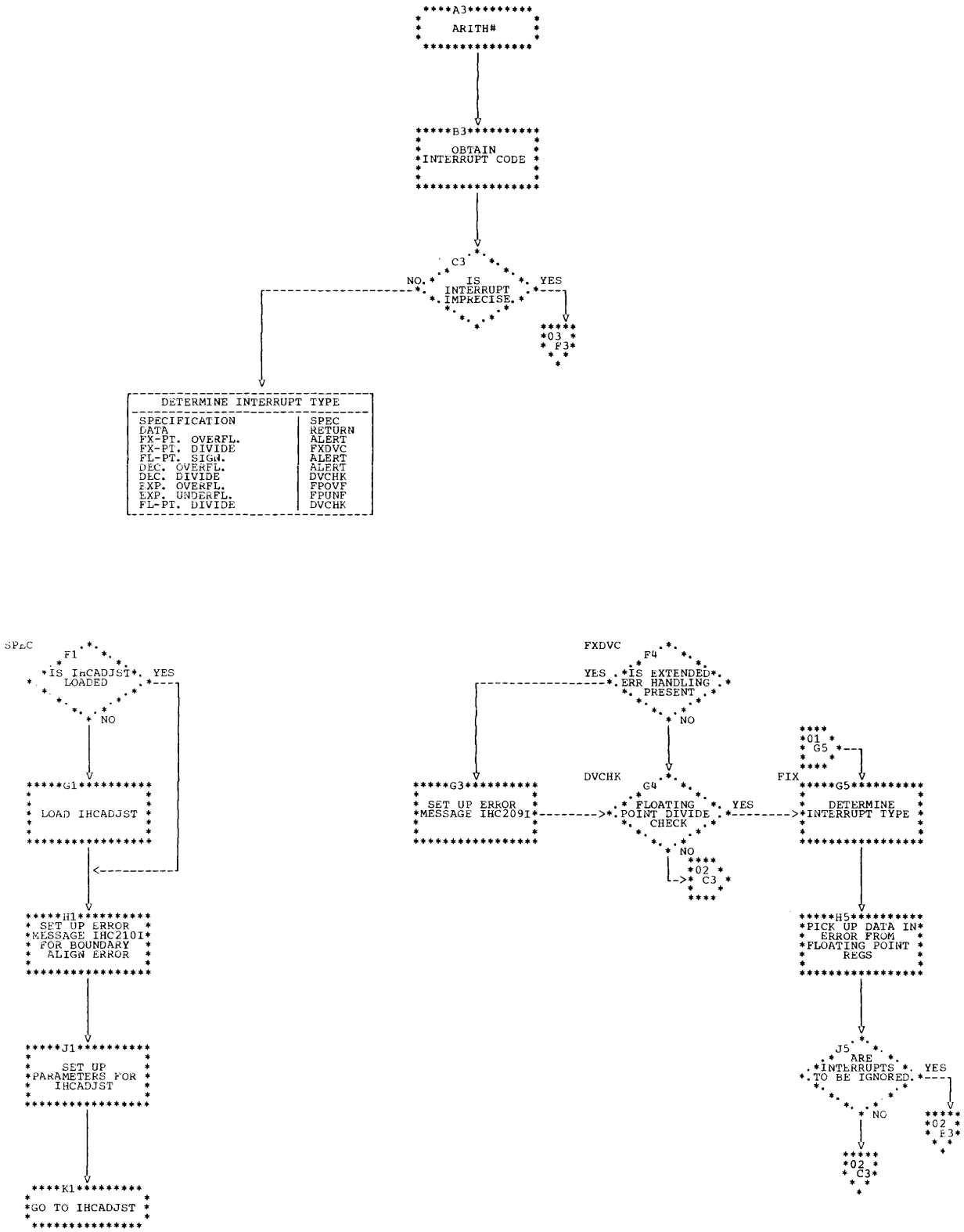


Chart 27. IHCFINTH/IHCEFNTH (Part 2 of 3)

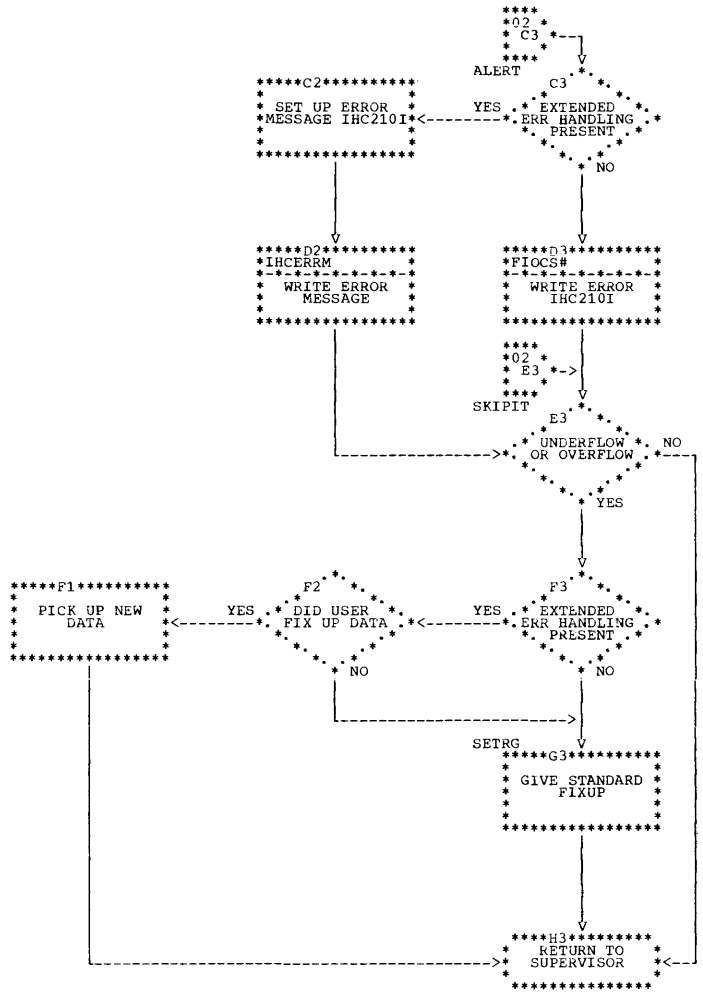


Chart 27. IHCFINTH/IHCEFNTH (Part 3 of 3)

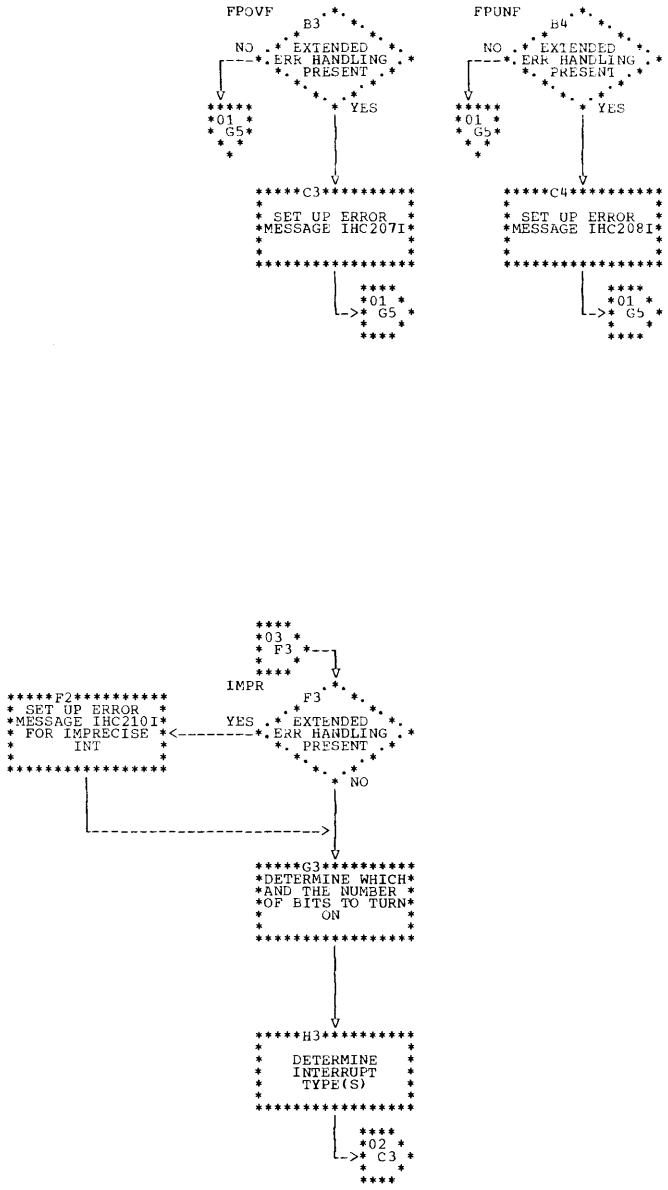


Chart 28. IHCADJST

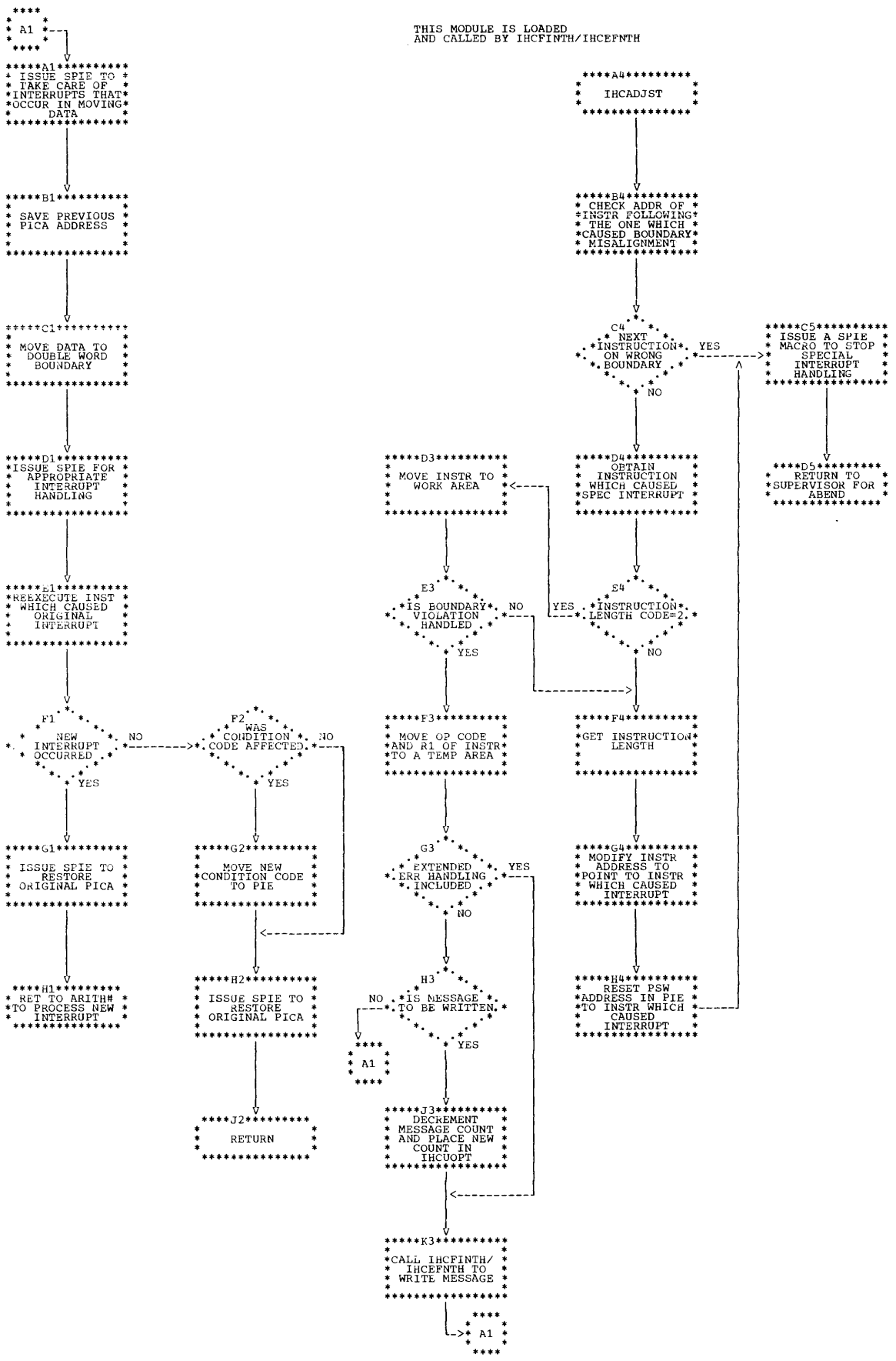


Chart 29. IHCIBERH

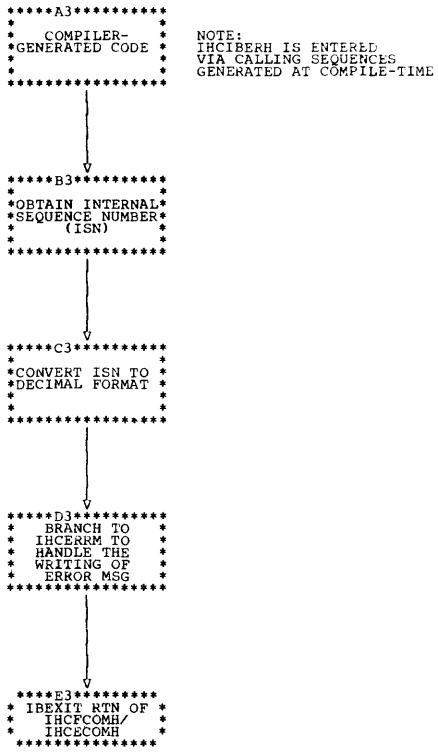


Chart 30. IHCSTAE (Part 1 of 2)

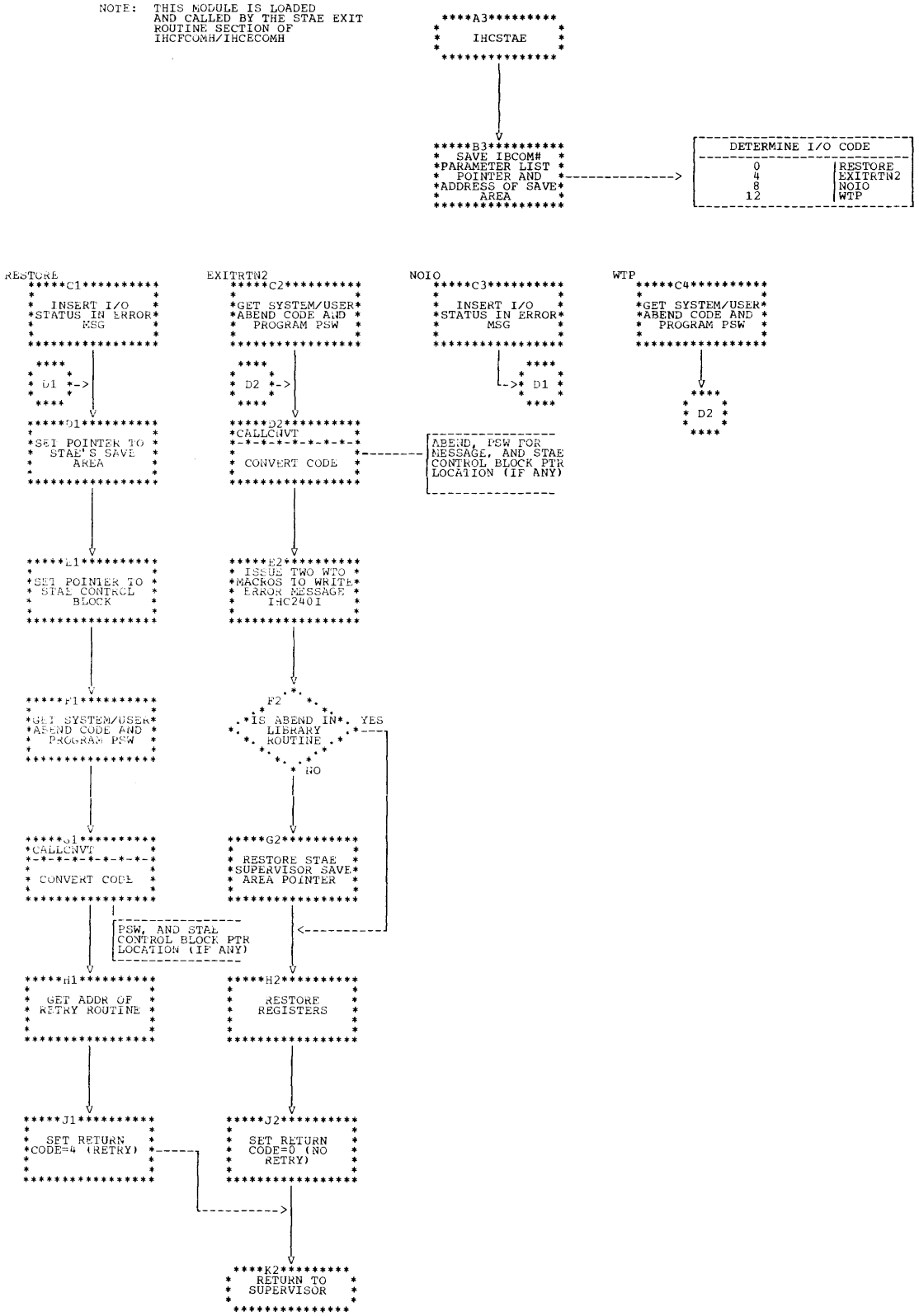


Chart 30. IHCSTAE (Part 2 of 2)

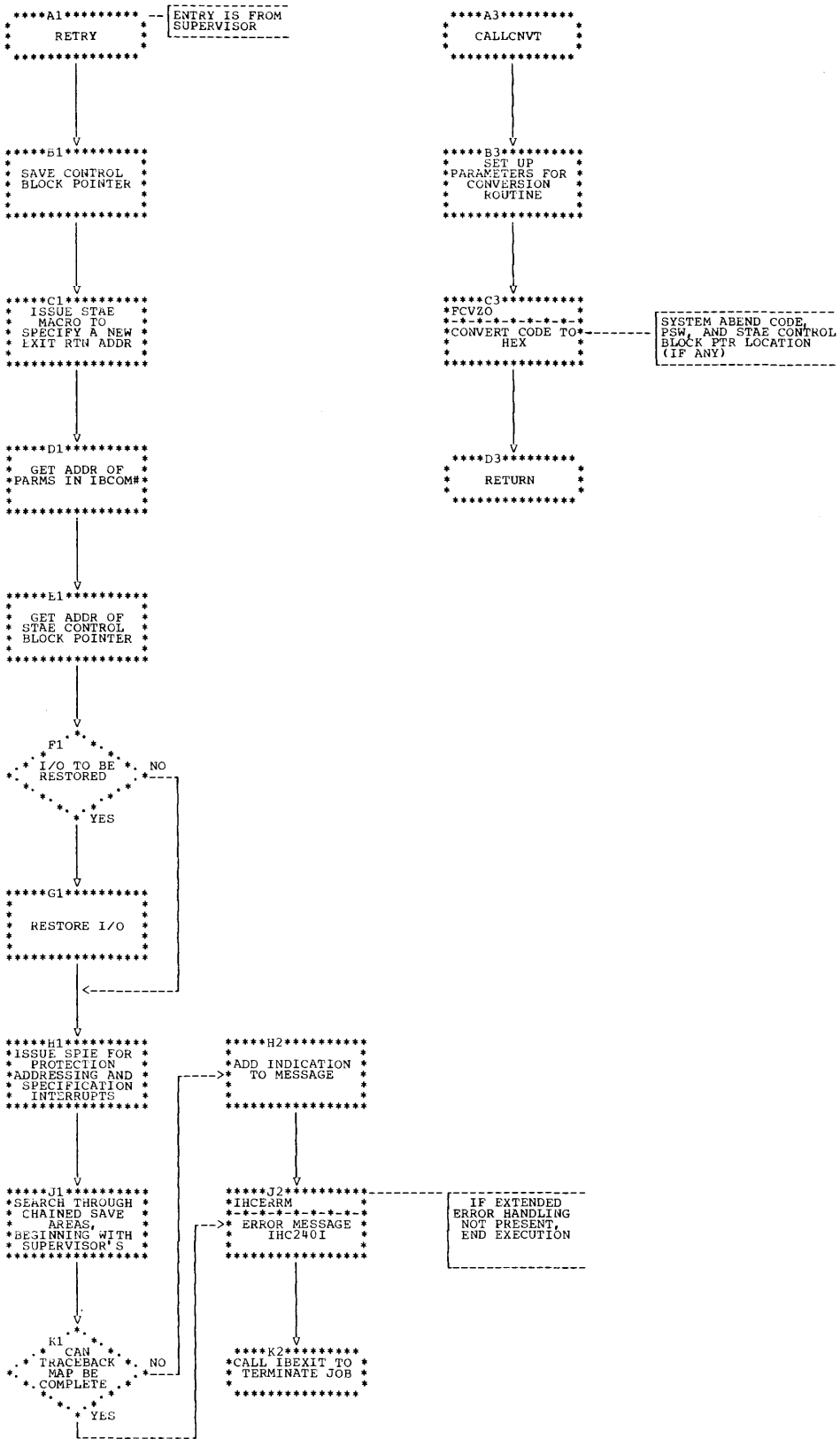


Chart 31. IHCERRM (Part 1 of 2)

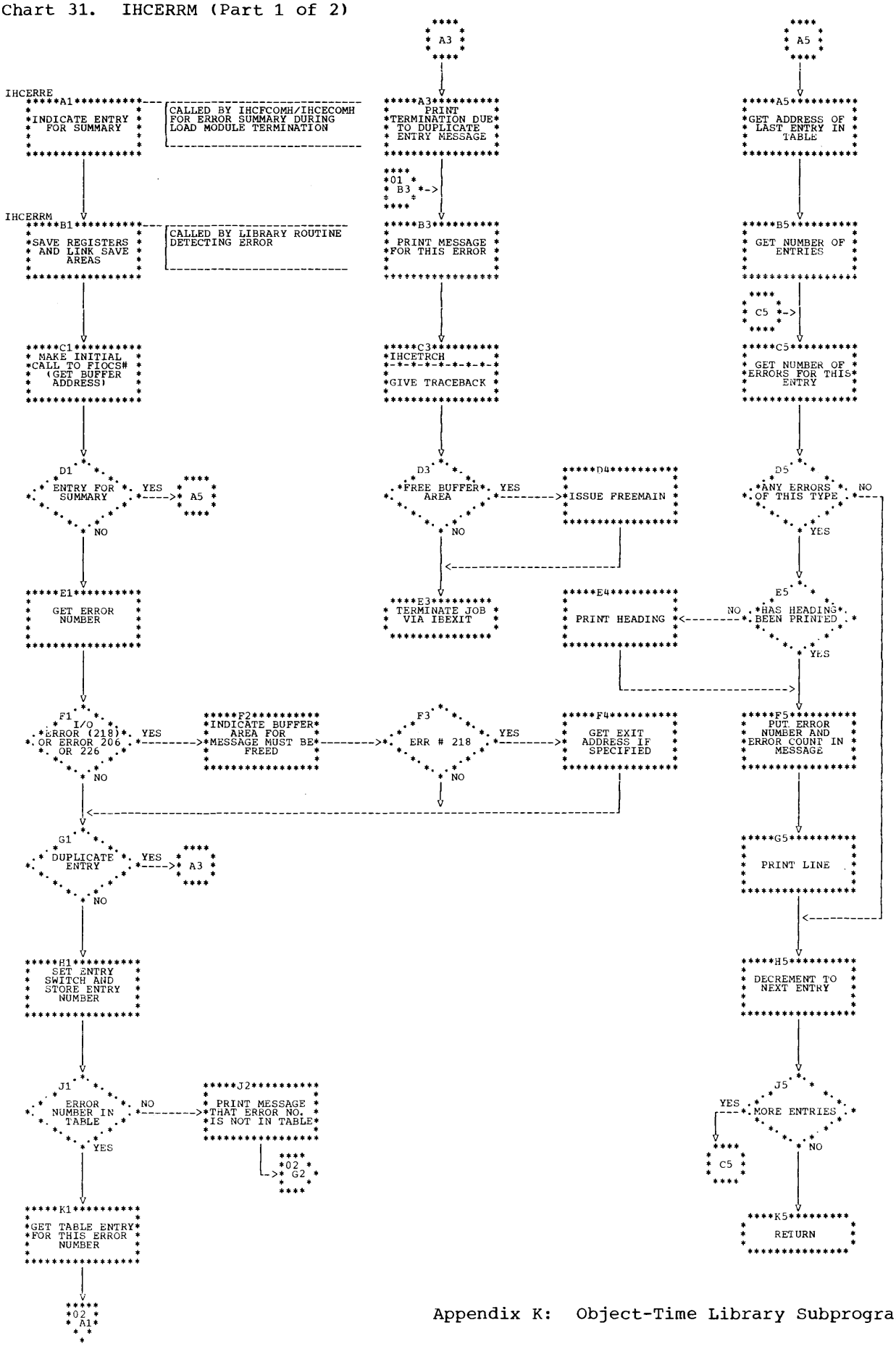


Chart 31. IHCERRM (Part 2 of 2)

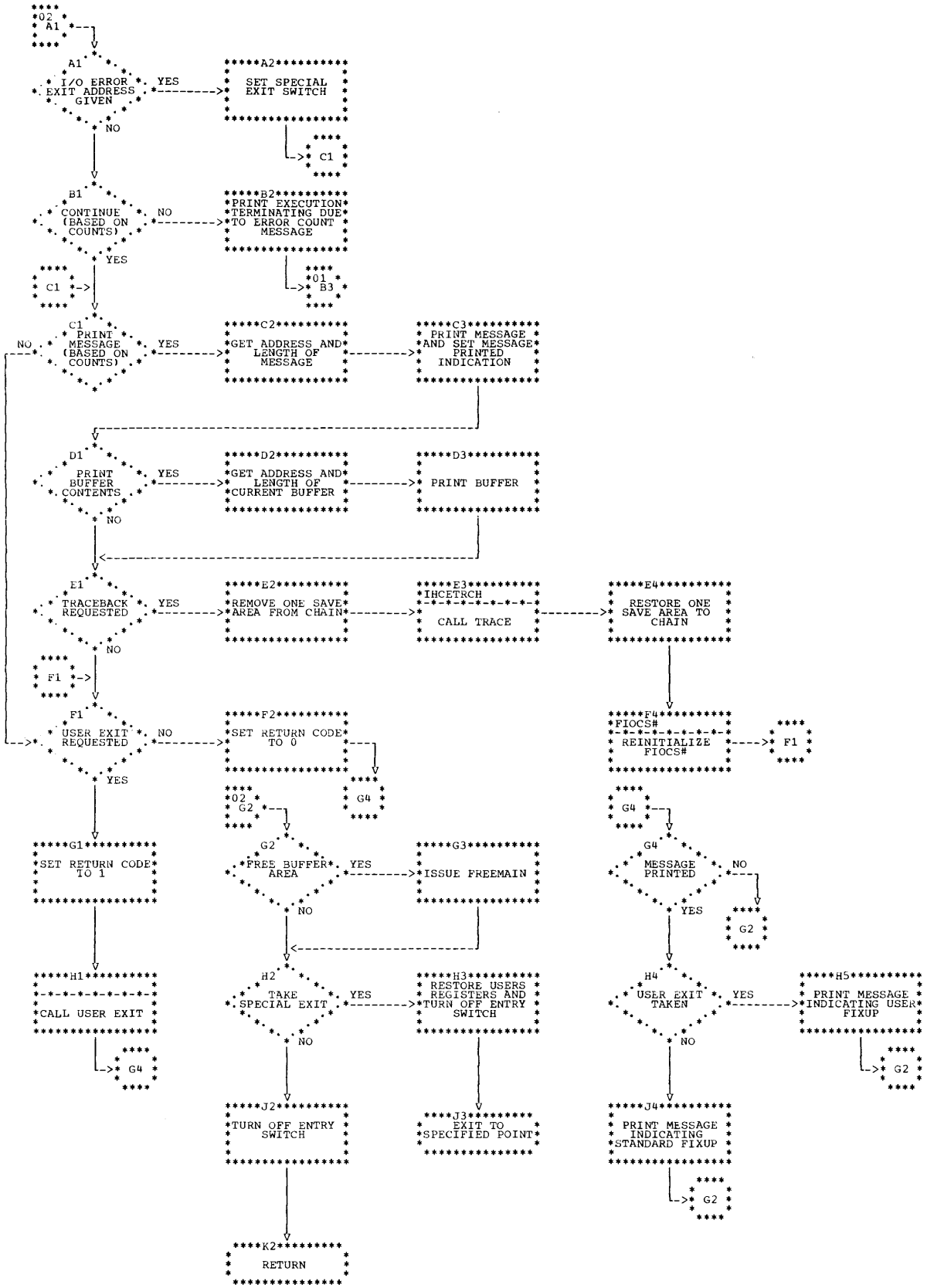


Chart 32. IHCFOPT (Part 1 of 3)

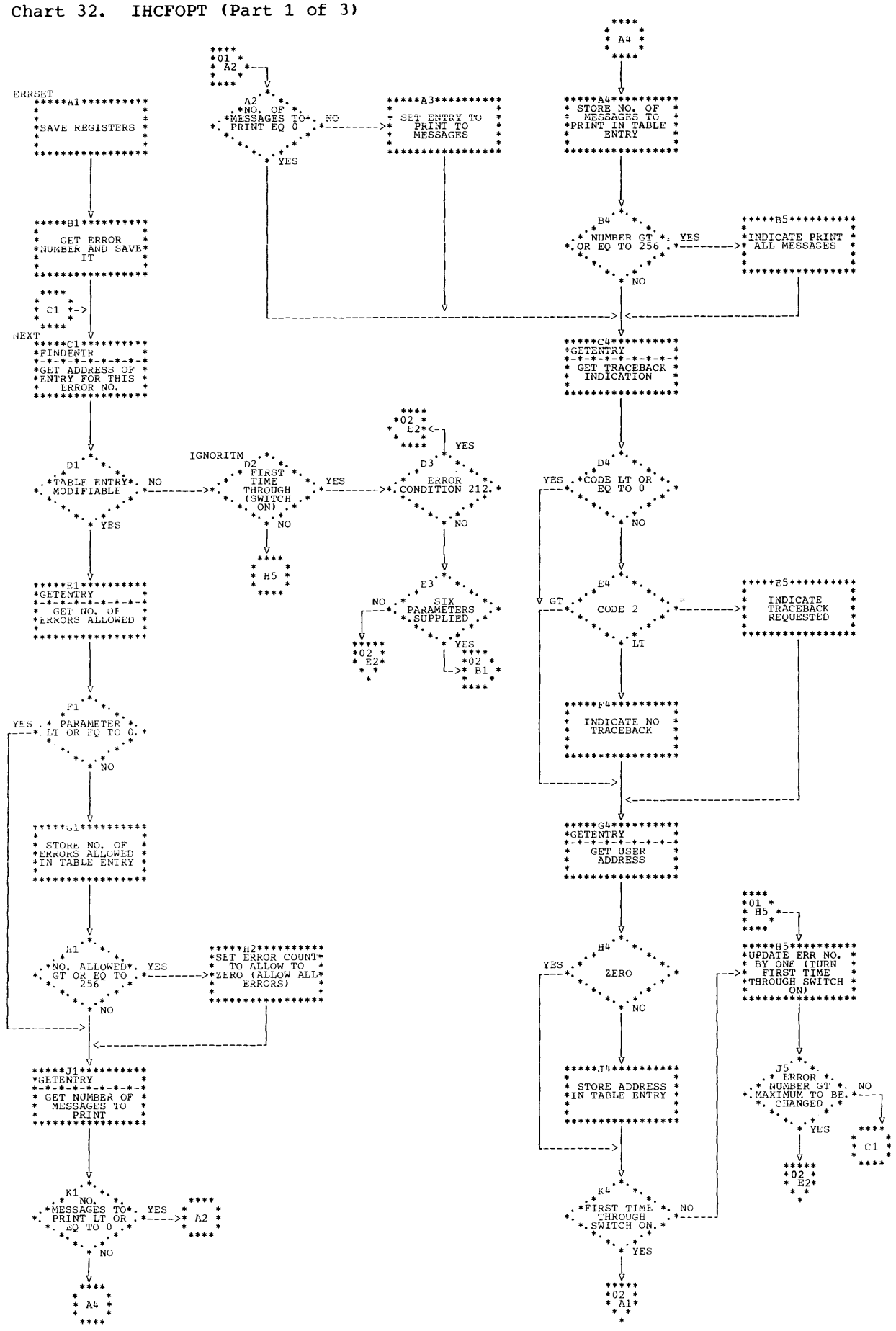


Chart 32. IHCFOPT (Part 2 of 3)

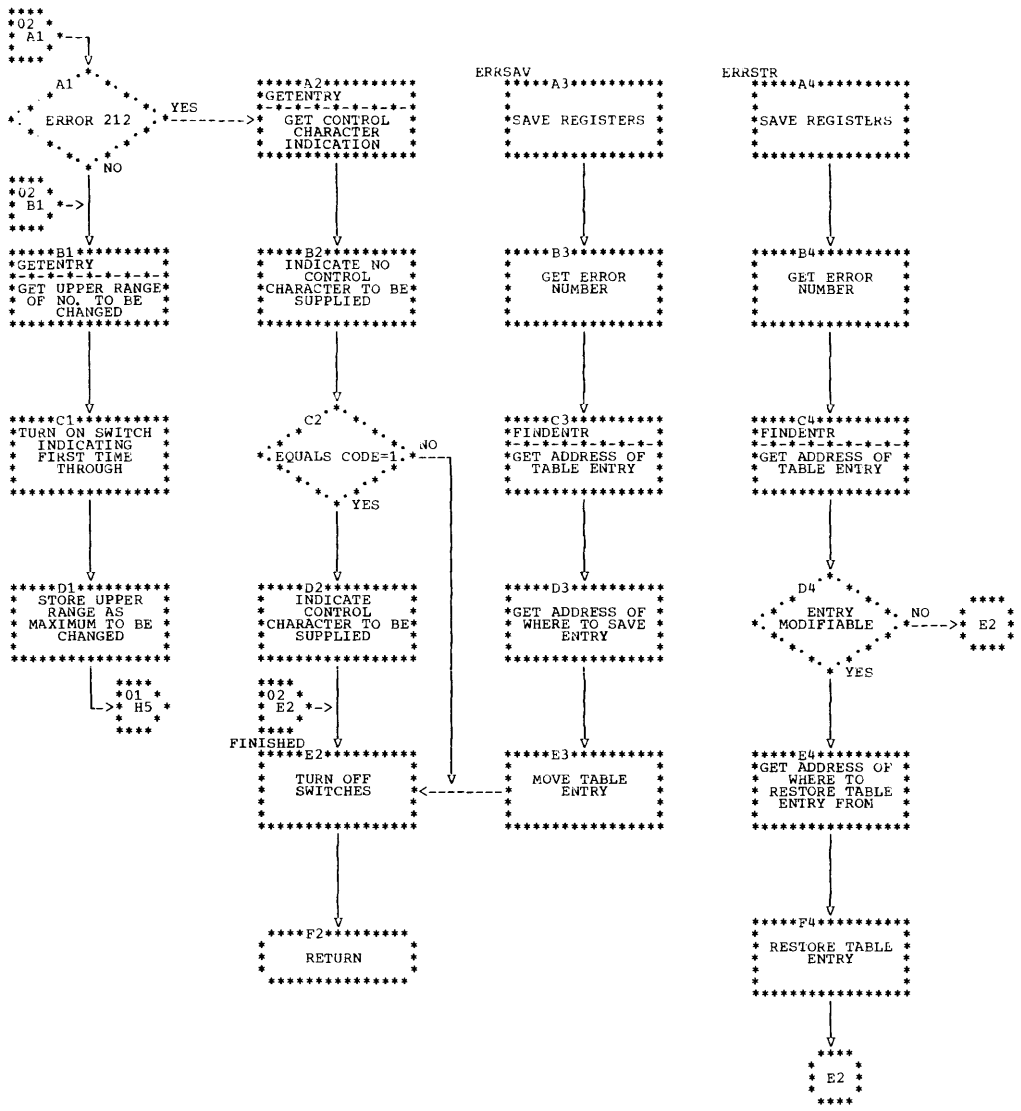


Chart 32. IHCFOPT (Part 3 of 3)

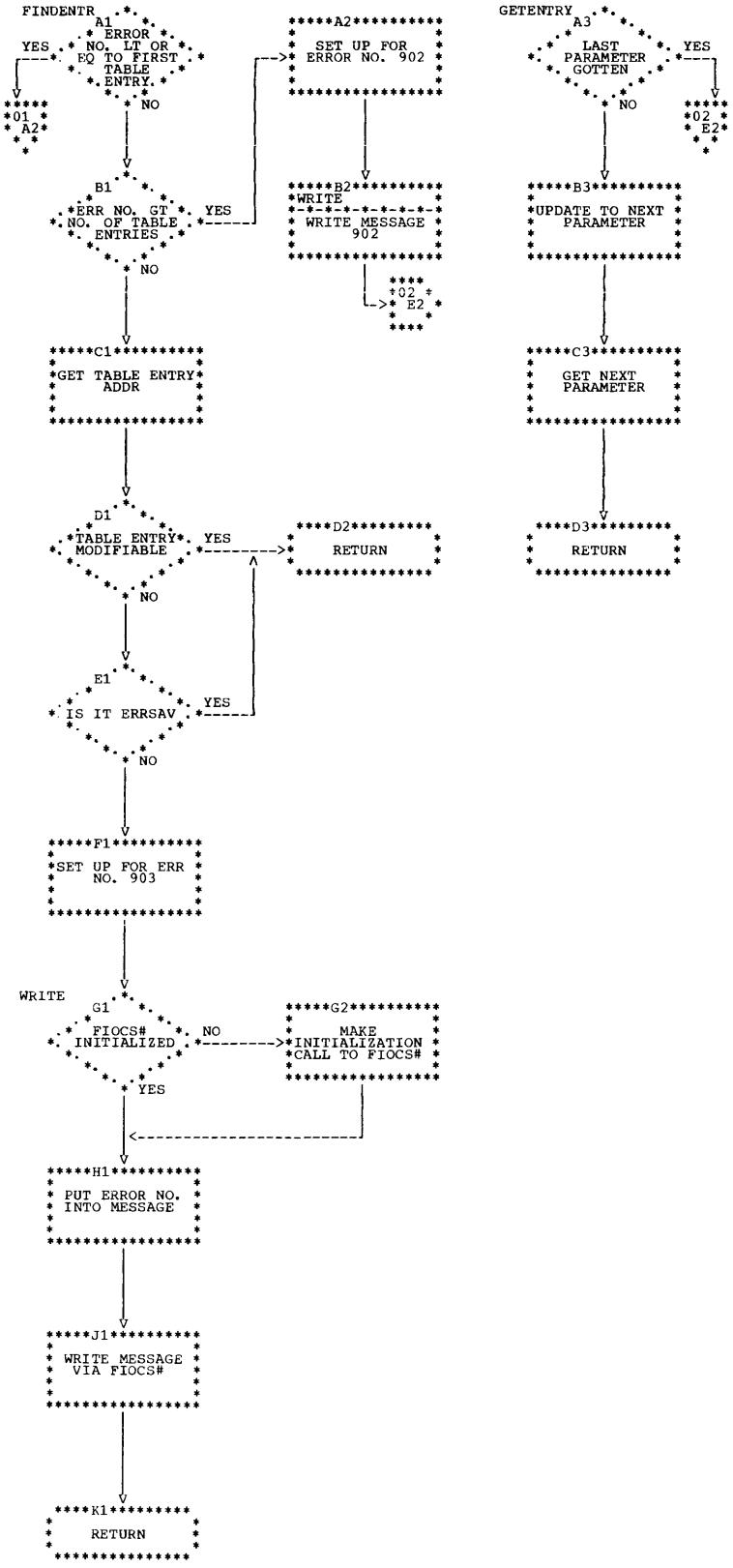


Chart 33. IHCTRCH/IHCERTCH

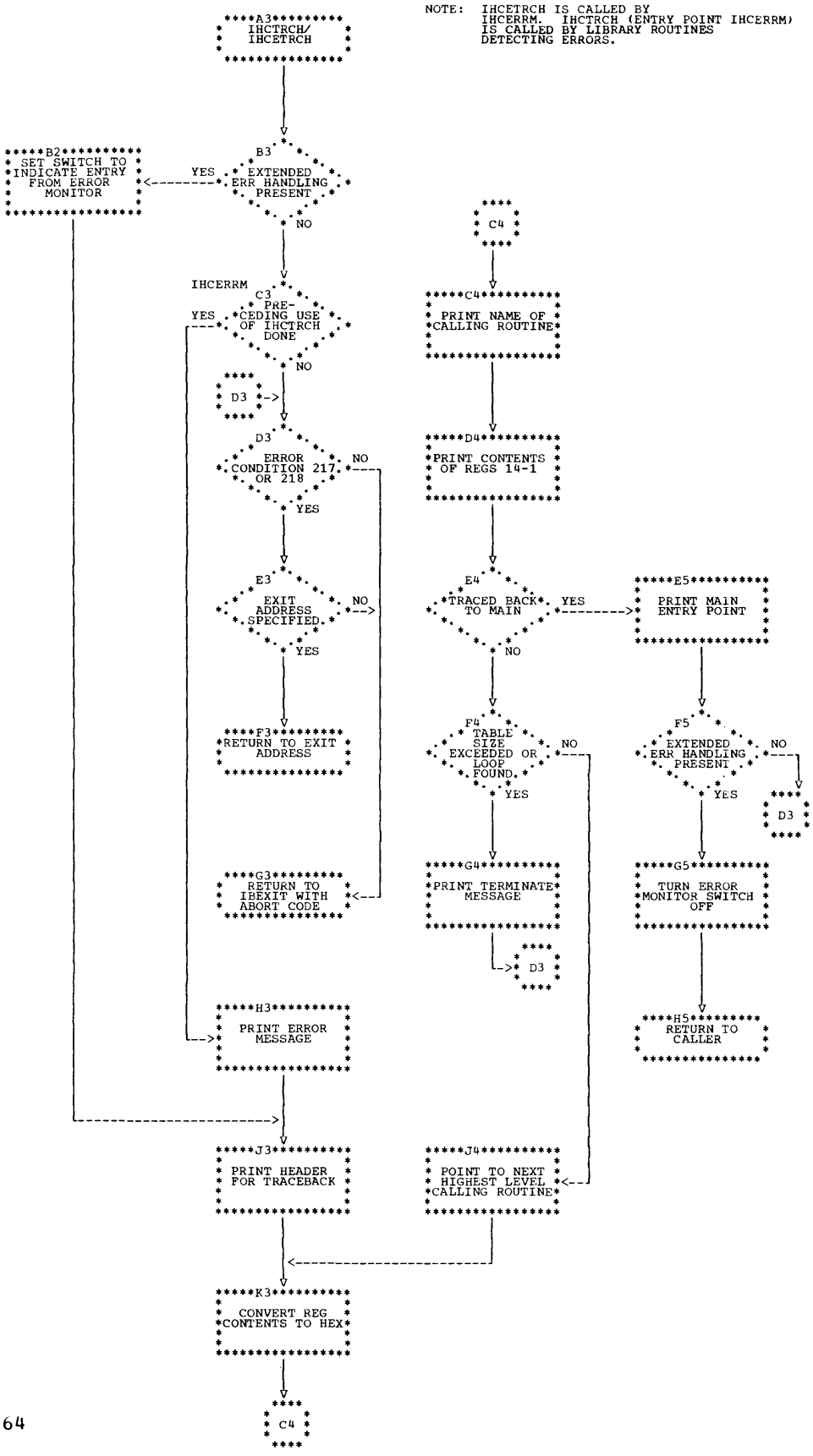


Chart 34. IHCFDUMP

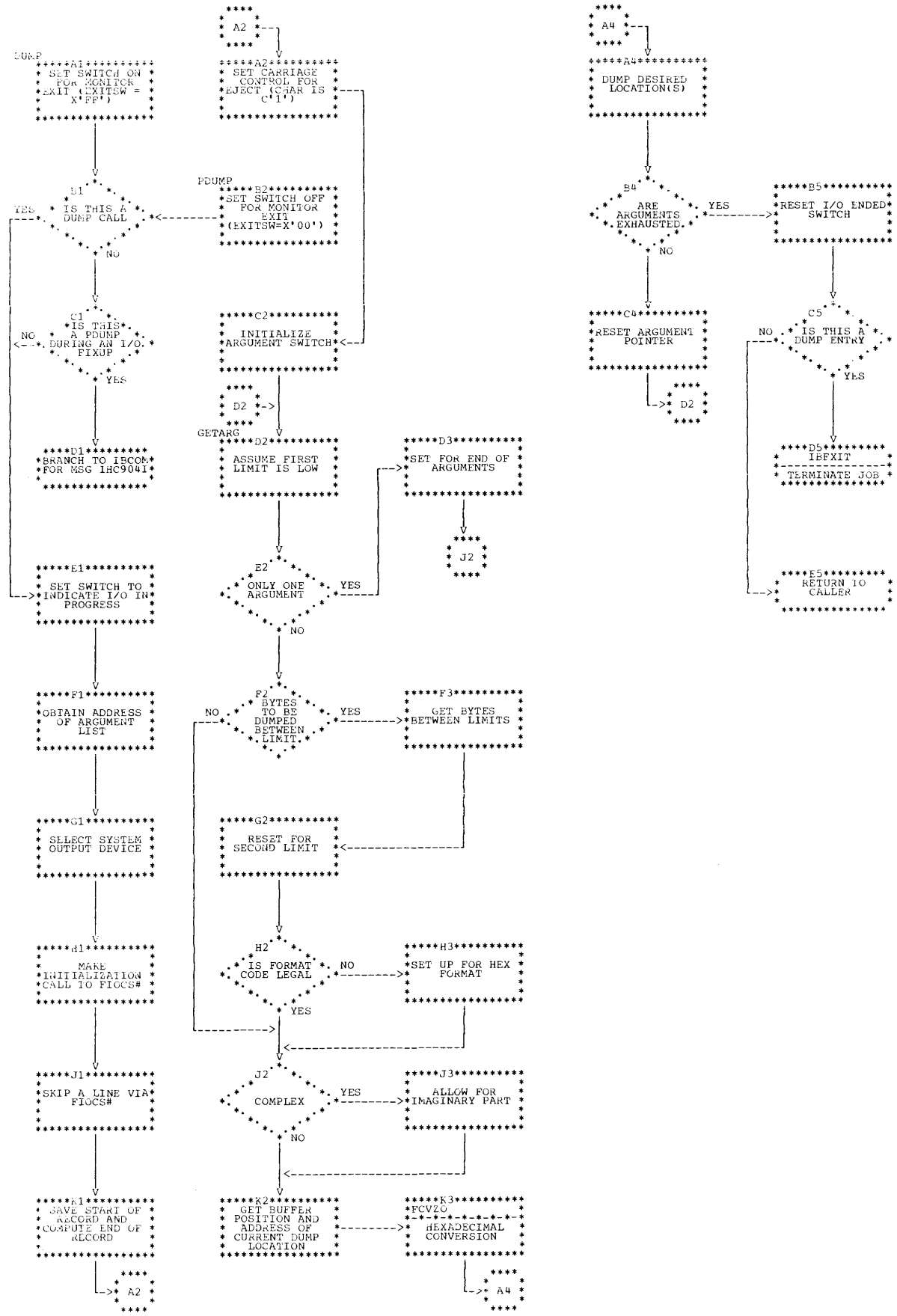


Chart 35. IHCFEXIT

Chart 36. IHCFSLIT

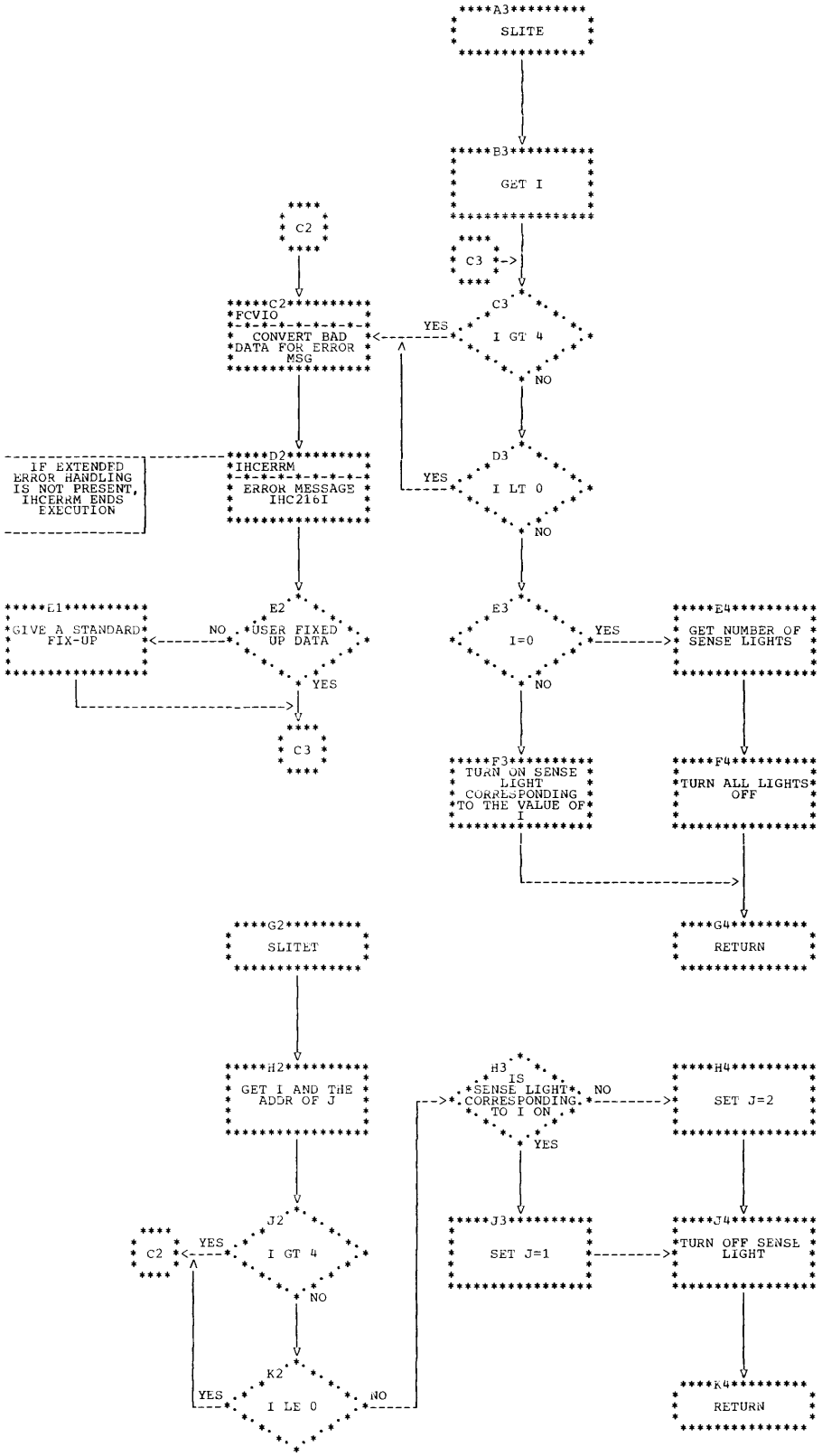


Chart 37. IHCFOVER

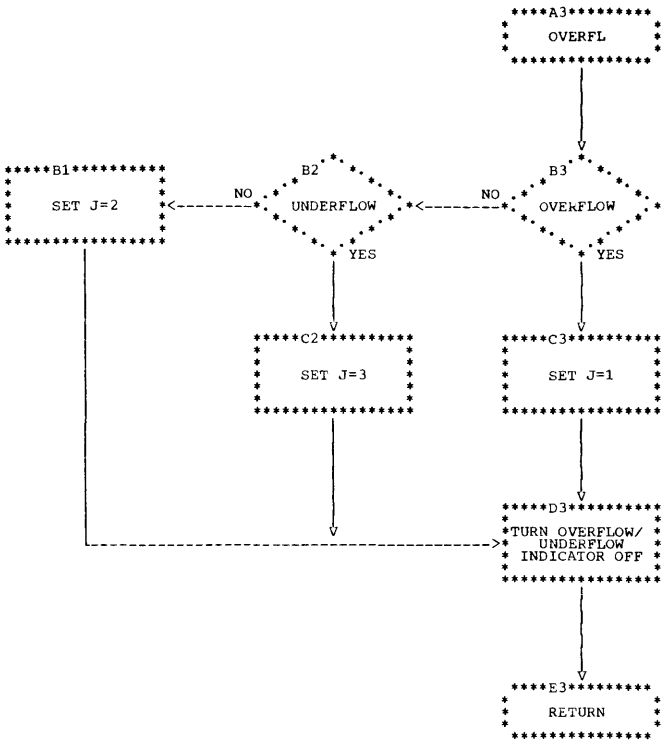
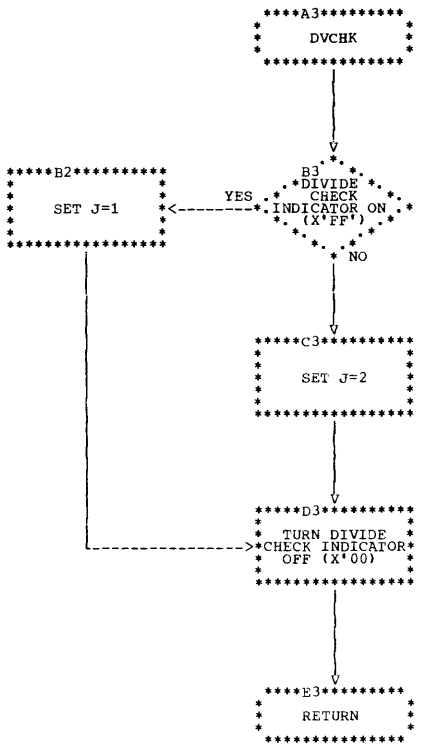


Chart 38. IHCFDVCH



Printed in U.S.A. GY28-6642-5

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, New York 10604
[U.S.A. only]

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
[International]