File No. S360-25
Form No. Y28-6642-1

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 which 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 produces optimized object modules (modules that can be executed with improved efficiency).

This program logic manual is directed to the IBM customer engineer who is responsible for program maintenance. It can be used to locate specific areas of the program and it enables the reader 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.

This revision reflects the 5.1 version of the FORTRAN IV (H) compiler program. A number of table formats and intermediate text formats have been changed. The overall operation of the compiler has not changed significantly, but some routines within the program have been changed, new routines have been added, and some routines have been deleted or combined with other routines.

Restricted Distribution

Second Edition (November 1967)

This publication corresponds to Release 14. It is a major revision of and makes obsolate, Form Y28-6642-0. New appendixes headed "Microfiche Directory" and "Facilities used by the Compiler" are added. Significant changes have been made throughout this publication to reflect changes in the program. New or modified material is indicated by a vertical line in the left-hand margin. The symbol • to the left of a caption indicates a revision to the illustration.

Specifications contained herein are subject to change from time to time. Any such change will be reported in subsequent revisions or Technical Newsletters.

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

A form for readers' comments appears at the back of this publication. It may be mailed directly to IBM. Address any additional comments concerning this publication to the IBM Corporation, Programming Systems Publications, Department D58, PO Box 390, Poughkeepsie, N. Y. 12602

This publication provides customer engineers and other technical personnel with 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, Form A22-6821

IBM System/360 Operating System:

FORTRAN IV, Form C28-6515

Introduction to Control Program Logic, Program Logic Manual, Form Y28-6605

FORTRAN IV (H) Programmer's Guide, Form C28-6602

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

IBM System/360 Operating System:

Sequential Access Methods, Program Logic Manual, Form Y28-6604

Concepts and facilities, Form C28-6535

Supervisor and Data Management Macro Instructions, Form C28-6647

Linkage Editor, Program Logic Manual, Form Y28-6610

System Generation, Form C28-6554

This manual consists of two parts:

1. An Introduction, describing 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.

2. A Body, containing a description of each component. Each component is discussed in terms of the functions it performs and 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 the second part are a number of appendixes, which contain descriptions of tables used by the compiler, intermediate text formats, a section on object-time library subprograms, the overlay structure of the compiler, 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.

CHARTS

TABLES

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 which 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.

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 (H) Programmer's Guide.

The compiler receives control from the calling program (e.g., job scheduler or another program that calls, links to, or attaches 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.)

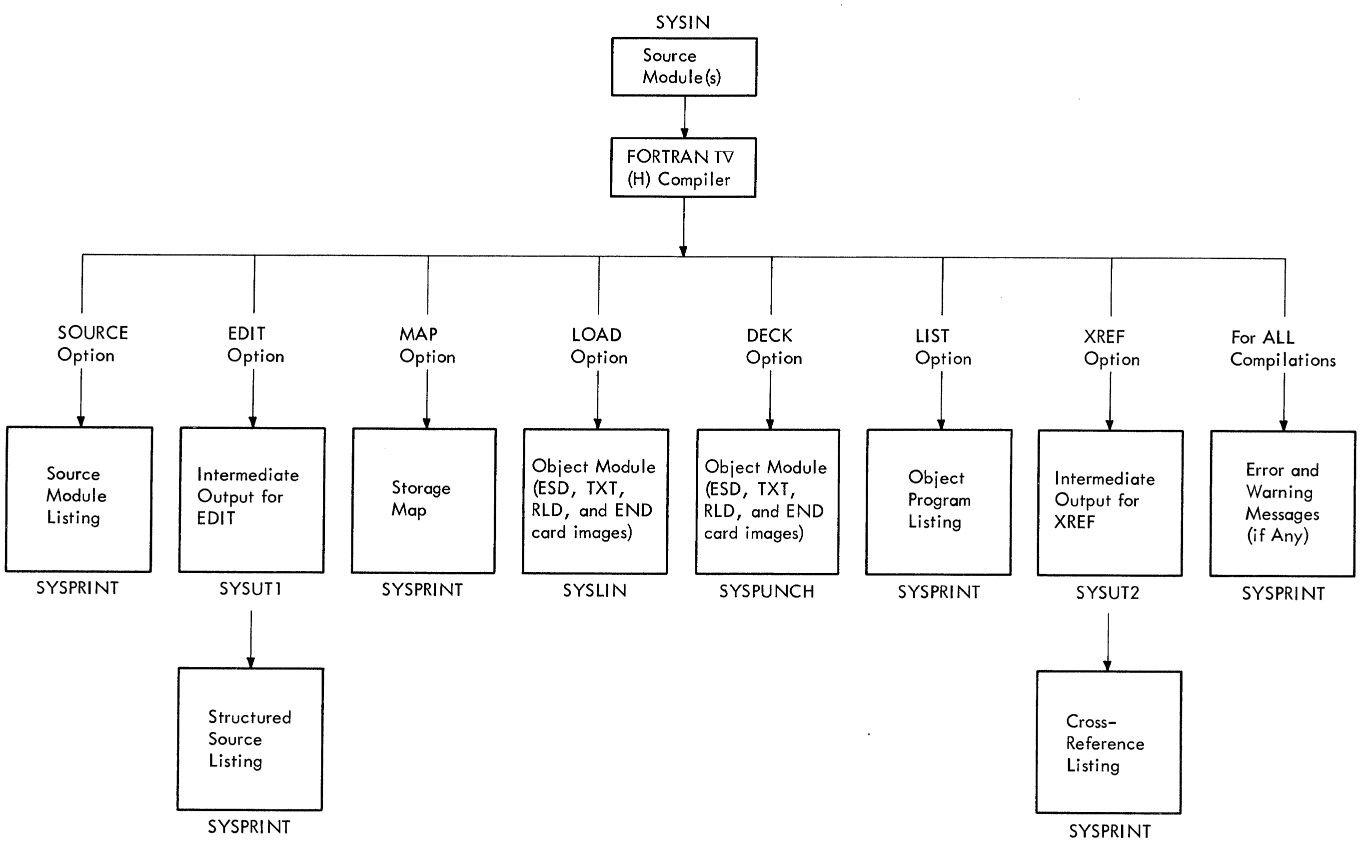
The components of the compiler operating together produce an object module from a FORTRAN source module. The object module is acceptable as input to the linkage editor, which prepares object modules for relocatable loading and execution.

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.

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.



• Figure 1. Input/Output Data Flow

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 in the information table about COMMON and EQUIVALENCE statements 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 form, 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 form, 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.

During processing, phase 20 records directly on the SYSPRINT data set messages describing any errors it detects and, if both the EDIT option and the second level of optimization are selected, produces, on the SYSPRINT data set, a structured source program listing.

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 form. 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. Messages for any errors detected during phase 25 processing are also recorded directly on SYSPRINT.

PHASE 30

Phase 30 is called after phase 15 processing is completed only if errors are detected by phases 10 or 15. Phase 30 records on the SYSPRINT data set messages describing the detected errors. Serious errors cause the compilation to be deleted before phase 20 processing begins.

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 G.

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 general operation.

FORTRAN SYSTEM DIRECTOR

The FORTRAN System Director (FSD) controls compiler processing; its overall logic is illustrated in Chart 01. 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 two steps:

- Parameter processing.
- 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 (refer to 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.

Data Field Initialization

Data field initialization is concerned with 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 (refer to 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 phase 10 or phase 15, phase 30 is called after the completion of phase 15 processing.

STORAGE DISTRIBUTION

Phases 10, 15, and 20 require main storage space in which to construct the information table (refer to 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 sub-block) 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 sub-block; refer to Appendix B, "Phase 10 Intermediate Text") pointers to both the beginning and end of the allocated storage space. If the sub-block is allocated for phase 10 normal text or for the information table, the pointers are returned in the communication table. If the sub-block 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 sub-block. This process is repeated for each phase 10 request for main storage space.

Sub-blocks 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 sub-block with text entries, it requests another.) A request for a sub-block to contain a particular type of entries may immediately follow a request for a sub-block to contain another type of entries. Consequently, sub-blocks allocated to contain the same type of entries may be scattered throughout main storage. The FSD must keep track of the sub-blocks 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 storage are passed to phase 15 via the communication table. Pointers to both the beginning and end of the allocated sub-block portion are passed to phase 15 via the communication table. If an additional request is received after the last sub-block 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 sub-blocks that were allocated to phase 10 normal text. These sub-blocks 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 sub-block allocated to phase 10 normal text is placed into that field. The chain connecting the various sub-blocks allocated to phase 10 normal text is then scanned and when a zero pointer field is encountered, a pointer to the first sub-block allocated to SF skeleton text is placed into that field. Once the sub-blocks 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 sub-blocks 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 sub-block allocated for phase 10 special text is placed into that field. After the sub-blocks 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 to 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 (refer to IBM System/360 Operating System: Sequential Access Methods, Program Logic Manual.)

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: I*w* (output only), T*w*, A*w*, *w*X, *w*H, and Z*w* (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 combination (refer to Appendix E) to implement sequential READ/WRITE statements requiring a format.

GENERATION OF INITIALIZATION INSTRUCTIONS

The FSD subroutine IEKTLOAD works with phase 25 to generate the machine instructions for entry into a main program, a subprogram, or a subprogram secondary entry point. These instructions are referred to as initialization instructions and are divided into three catagories:

- Main program entry coding.
- Subprogram main entry coding.
- Subprogram secondary entry coding.

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

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

- Save the contents of general registers 14 through 12.
- Load the reserved registers with their associated addresses. (The address loaded into register 13 is that of the save area. The address loaded into register 11, if reserved, is that of the save area plus 4096 bytes. The address loaded into register 10, if reserved, is that of the save area plus 8192 bytes. The address loaded into register 9, if reserved, is that of the save area plus 12,288 bytes.)
- Load the address of the main program save area into register 4, and store register 4 into the save area of the calling program.
- Save register 13 in the new save area.

Subprogram Main Entry Coding: The initialization instructions generated by subroutine IEKTLOAD for the main entry point into a subprogram perform the following functions:

- Save the contents of general registers 14 through 12.
- Load the addresses of the prologue and epilogue of the subprogram into registers. (For an explanation of prologue and epilogue, refer to "Phase 25, Prologue and Epilogue Generation.")
- Load the reserved registers with their associated addresses.
- Load the address of the save area of the subprogram into register 13.

* Save the address of the save area of the calling routine and the address of the epilogue of the subprogram in the save area of the subprogram.

* Branch to the prologue.

* Set up a save area in which the contents of the registers used by the subprogram are saved, should that subprogram, in turn, call another subprogram.

* Set up address constants in which the addresses of the prologue and epilogue of the subprogram and the addresses to be placed into the reserved registers are inserted.

Subprogram Secondary Entry Coding: The initialization instructions for a subprogram secondary entry point are essentially the same as those required for the main entry point. For this reason, IEKTLOAD makes use of a number of the initialization instructions for the main entry point in processing secondary entry points.

Main entry point initialization instructions that precede and include the instruction that loads the prologue and epilogue addresses cannot be used, because each secondary entry point has its own associated prologue and epilogue. Therefore, for secondary entry points, subroutine IEKTLOAD generates initialization instructions that perform the following functions:

* Save the contents of general registers 14 through 12.

* Load the addresses of the prologue and epilogue of the secondary entry point into registers.

* Branch to the subprogram main entry point initialization instruction that loads the reserved registers with their associated addresses.

* Set up address constants in which the addresses of the prologue and epilogue of the secondary entry point are placed.

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 initialization instructions for secondary entry points are generated when the text representation of an ENTRY statement is encountered during the last scan of intermediate text. These instructions reside in the "Instructions" section of text information.

DELETION OF A COMPILATION

The FSD deletes a compilation if either of the following occurs:

* An error of error level code 16 (refer to the publication IBM System/360 Operating System: FORTRAN IV (H) Programmer's Guide) is detected during the execution of a processing phase.

* The value of the error level code returned from phase 30 is 8 and the LOAD option has not been specified.

In the former case, 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 processing 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.

In the latter case, phase 30 returns control to the FSD at the next sequential instruction. If the error level code passed to the FSD is 8 and the LOAD option has not been specified, the FSD continues processing.

Note: Phase 25 returns an error level code of 8 to the FSD if errors are detected during the translation of FORMAT statements. However, in this case, the FSD does not delete the compilation if the LOAD option has not been specified.

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.

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:

* 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 appearing 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 what 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. After the entire source statement has been processed, the next 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 calls phase 15 for execution.

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

- Dispatcher subroutine
- Preparatory subroutine
- Keyword subroutines
- Arithmetic subroutines
- Utility subroutines
- STALL-IEKGST subroutine

Dispatcher Subroutine

The dispatcher subroutine (DSPTCH-IEKCDP) controls phase 10 processing. Upon receiving control from the FSD, 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 if a statement number is associated with the source statement being processed. If there is a statement number, the DSPTCH-IEKCDP subroutine constructs a statement number entry (refer to Appendix A, "Information Table") for the statement number. A text entry for the statement number is also created. The DSPTCH-IEKCDP subroutine then determines, from the classification code assigned to the source statement (refer to "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, after being prepared, resides in the storage area NCARD/NCDIN in the format illustrated in Figure 2.

NCARD	
Pointer to first character of packed source statement beyond keyword[1]	(1 word)
Internal statement number	(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.

Figure 2. Format of Prepared Source Statement

----

[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.

Keyword Subroutines

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 (refer to Table 8). It is associated with a COMMON, DIMENSION, EQUIVALENCE, or EXTERNAL keyword statement.

This subroutine scans its associated statement (in NCDIN) in a left-to-right fashion 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 this subroutine 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 (refer to 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 sub-block 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 Subroutines

The arithmetic subroutines (refer to Table 8) receive control from the DSPTCH-IEKCDP subroutine, or from various keyword subroutines, and 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.)

The following paragraphs describe the processing of the arithmetic subroutines according to their functions.

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, a zero (null) operand is assumed.

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 if 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, 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 Subroutines

The utility subroutines (refer to 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., sub-blocks) 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 DO's appearing in I/O 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

STALL-IEKGST 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 offset (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 IHCFCOMH. STALL-IEKGST calls subroutine FORMAT-IEKTFM which develops the necessary form 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. FORMAT-IEKTFM 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 constant for the statement number associated with the FORMAT statement.

STALL-IEKGST reserves storage within text information for the variables and arrays of the module between the last constant and the first translated FORMAT statement, or the first object-time namelist dictionary, if FORMAT statements do not exist in the module. To accomplish this, STALL-IEKGST assigns to the first translated FORMAT statement (or object-time namelist dictionary) a relative address equal to the number of bytes occupied by the constants, variables, and arrays of the module.

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

Processing Literal Data Text: STALL-IEKGST checks a pointer in the communication table (NPTR (1,27)) to see if there are literal constants to process. If there are, STALL-IEKGST calls IEKTLOAD and passes it the location and length of the literal string which IEKTLOAD uses to generate literal data text in the object module.

STALL-IEKGST follows the chain in the literal constant dictionary entry and continues to call IEKTLOAD to process this text. After all the literal data text has been generated, STALL-IEKGST adjusts the relative object location counter by the amount of text generated.

Reserving Space for the Save Area: STALL-IEKGST sets aside space for the save area of the program being compiled. The amount of space reserved depends on the type of program being processed. For a program with no external CALLs, 16 bytes are required for the save area. A program with external CALLs needs a save area 76 bytes long.

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

•Table 1. FORMAT Statement Translation

FORMAT Specification	Description	Translated Form (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.

Checking for Undefined Statement Numbers: STALL-IEKGST 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. STALL-IEKGST scans the chain of statement number entries in the information table (refer to 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, STALL-IEKGST places an entry in the error table for later processing by phase 30.

Rechaining Entries for Variables: STALL-IEKGST 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). STALL-IEKGST 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. STALL-IEKGST also constructs dictionary entries for the imaginary parts of complex variables and constants.

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

- The first 59 unique variables and/or constants appearing 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 (refer to CORAL Processing, "Adcon and Base Variable Assignment").

[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.

Subroutine IEKKOS assigns 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. IEKKOS assigns the second variable or constant a coordinate number of 3 and records its usage information in bit position 3 of the three fields. IEKKOS continues this process until coordinate 60 has been assigned to a variable or constant. After coordinate number 60 has been assigned, IEKKOS only assigns coordinates to the next 20 unique variables. 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. IEKKOS 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 (refer to Appendix A, "Dictionary") to assign, keep track of, and record coordinate numbers. MCOORD 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 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: STALL-IEKGST processes common entries in the information table. It computes the offsets (displacements) of variables and arrays from the start of the common block containing them and calculates the total size in bytes of each common block. STALL-IEKGST records the offsets in the dictionary entries for the variables and the block size in the common table entry for the name of the common block. The offsets 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. (Refer to Appendix A: "Common Table.") STALL-IEKGST 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: STALL-IEKGST gathers additional information about equivalence groups and the variables in them. It computes a group head[1] and the offset (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. (Refer to Appendix A: "Common Table".) 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, STALL-IEKGST 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, STALL-IEKGST recognizes the combination of groups and modifies the dictionary entries for the variables to indicate the equivalence operations. STALL-IEKGST checks arrays appearing 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, 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.

LABTLU-IEKCLT 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:

←——— 4 bytes ———→

Pointer to next XREF entry*	ISN

* Relative to the beginning of the buffer.

Each time the buffer is full, LABTLU-IEKCLT calls IEKXRS to write the buffer on SYSUT2. (The contents of SYSUT2 is later read in by 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, IEKXRS returns control to LABTLU-IEKCLT.

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 XREF-IEKXRF can create a sequential list of statement numbers used in the program.

The words used by LABTLU-IEKCLT 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 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 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.

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

CSORN-IEKCCR 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 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 XREF-IEKXRF after the completion of STALL-IEKGST processing and before phase

15. XREF-IEKXRF is a separate overlay segment that overlays phase 10 and is overlaid by phase 15.

XREF-IEKXRF reads from SYSUT2 all buffers that were written out by IEKXRS during LABTLU-IEKCLT and 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 which 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.

PHAZ15 translates and reorders the text entries for arithmetic expressions from the operator-operand format of phase 10 to a four-part form 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 four 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 100 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 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 OPT=2 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

----

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

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 order of statement numbers, as phase 10 recorded them, was in order of their occurrence in source statements as either definitions[1] or operands. The new sequence after phase 15 reordering is according to source statement occurrence as definitions only. The new order is established to facilitate phase 20 processing.

Lastly, 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.

Charts 06, 07, and 08 depict the flow of control during PHAZ15 execution.

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 the next 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. The text entries for each ENTRY statement therefore form a separate text block, which is referred to as an entry block.

Figure 3 illustrates the concept of text blocking. In the figure, 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 order in which algebraic operations are performed. Arithmetic expressions may exist in IF, CALL, and ASSIGN statements and I/O data-list, 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,

----

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

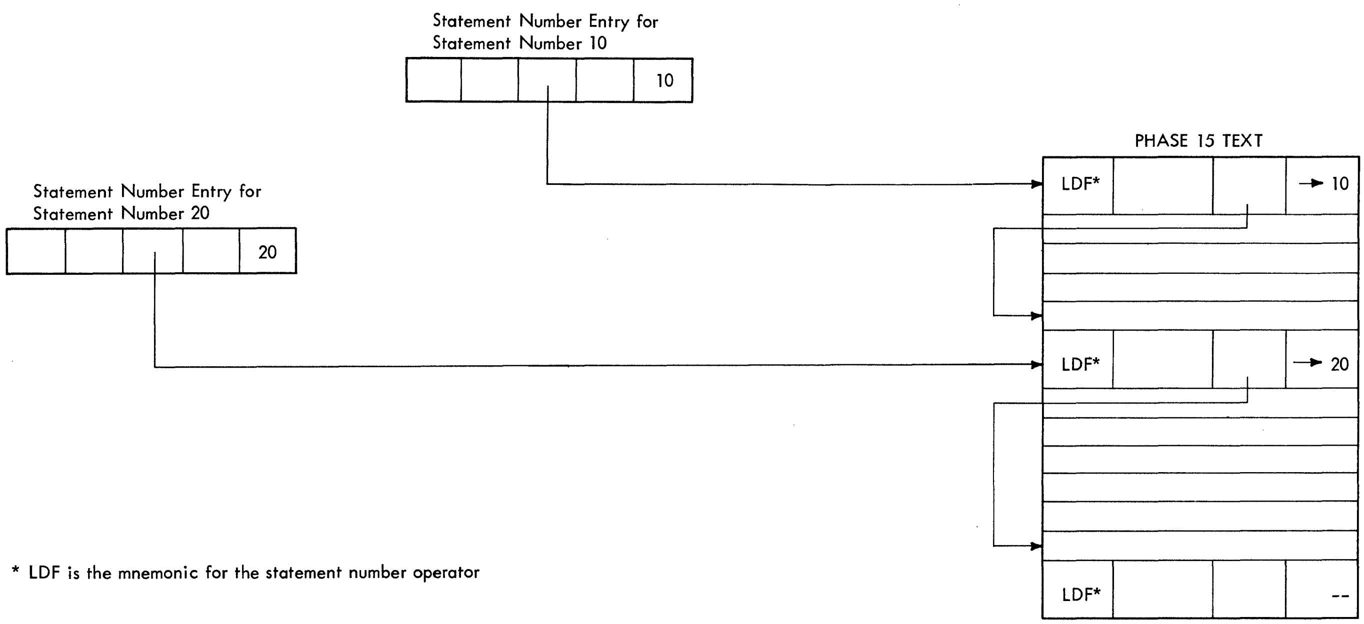


Figure 3. Text Blocking

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.

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 or whether a triplet is to be taken from the pushdown 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.

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

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.

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 figure) with its operand (operand B) and the operand of the next lower table entry (operand A). Note that 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). ALTRAN-IEKJAL assigns the previously generated temporary t as the operand of this pair. This temporary represents the previous operation (operator B--operand A--operand B).

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).

----

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

1. Text in Pushdown Table

	Operator	Operand
Top Element	Op B	Oprnd B
	Op A	Oprnd A

2. Phase 10 Text Entries

Operator	Operand	
Op C	Oprnd C	Current phase 10 text entry
Op D	Oprnd D	Next phase 10 text entry

4. New Top Element of Pushdown

Op A	t

3. New Phase 15 Text Entry

Op B	t	Oprnd A	Oprnd B
Operator	Operand 1	Operand 2	Operand 3

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

When it has been determined that special handling is needed, control is passed to one or more other subroutines (refer to 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 powers, 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

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. Lastly, a constant or variable raised to a constant integer power is converted to a series of multiplications (and a division into one, if necessary). 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 addition or multiplication), 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. (Refer to 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 subprogram table (IEKLFT) (refer to Appendix A, "Subprogram Table"), it is determined if the required routine is inline. Then the mode is tested. If the routine is in-line and the mode is as expected, 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 and/or number of arguments in an in-line function is not as expected, the function is assumed to be external.

If the mode and/or number of arguments in a library function is not as expected, another test is performed. The test determines if 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 subprogram table), text is generated to load the addresses of any arguments that are subscripted variables into a parameter list in the adcon table. (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.

If a function is not external, is in the subprogram 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 in the adcon table. (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 of 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 nonzero zero parameters in the list, a pointer to the adcon table (refer to 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:

Number of parameters	Number of nonzero parameters	NADCON Table pointer	Pointer to next entry of like format
20	16		
20	16		
10	7		
30	25		
20	16		
10	7		
20	16		
30	25		

---

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

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.

<u>Expressions Containing Statement Function References</u>: 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.

<u>Logical Expressions</u>: 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 if 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.

<u>Gathering Constant/Variable Usage Information</u>

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 optimized path through phase 20 (refer to "Phase 20"). If optimized 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 (refer to 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 (refer to CORAL PROCESSING, "Adcon and Base Variable Assignment") 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. 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 on, indicating that the operand is used in the block. 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, MATE-IEKLMA assigns it the next and records the following usage information for operand 1:

- Its associated coordinate bit in the MVX field is set 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 encountered in the block as a use and therefore is <u>not</u> not busy-on-entry.)
- Its associated coordinate bit in the MVS field is set on, indicating that it is defined within the block.

This process is repeated for all 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 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.

<u>Gathering Forward Connection Information</u>

An integral part of the processing of PHAZ15 is the gathering of forward connection information, which indicates which text blocks pass control to which other 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 (refer to "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 (refer to Appendix A, "Statement Number/ Array Table"). The statement number entry associated with the text block therefore 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 if it is a GO TO or compiler generated branch. If it is either, a pointer to the statement number entry for each statement number that may be branched to 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 text following a statement number definition and before the next, only one entry is made in RMAJOR for the statement number to be branched to.)

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 figure, 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 have been completed, subroutine PHAZ15-IEKJA reorders the statement number chain of the information table (refer to Appendix A, "Information Table"). The original order 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 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 order in which statement numbers are defined in the module, is built. The order of the entries in this table also reflects the order of the text blocks of the module.

After the scan, PHAZ15-IEKJA uses this table to reorder the statement number entries. It places the first table pointer

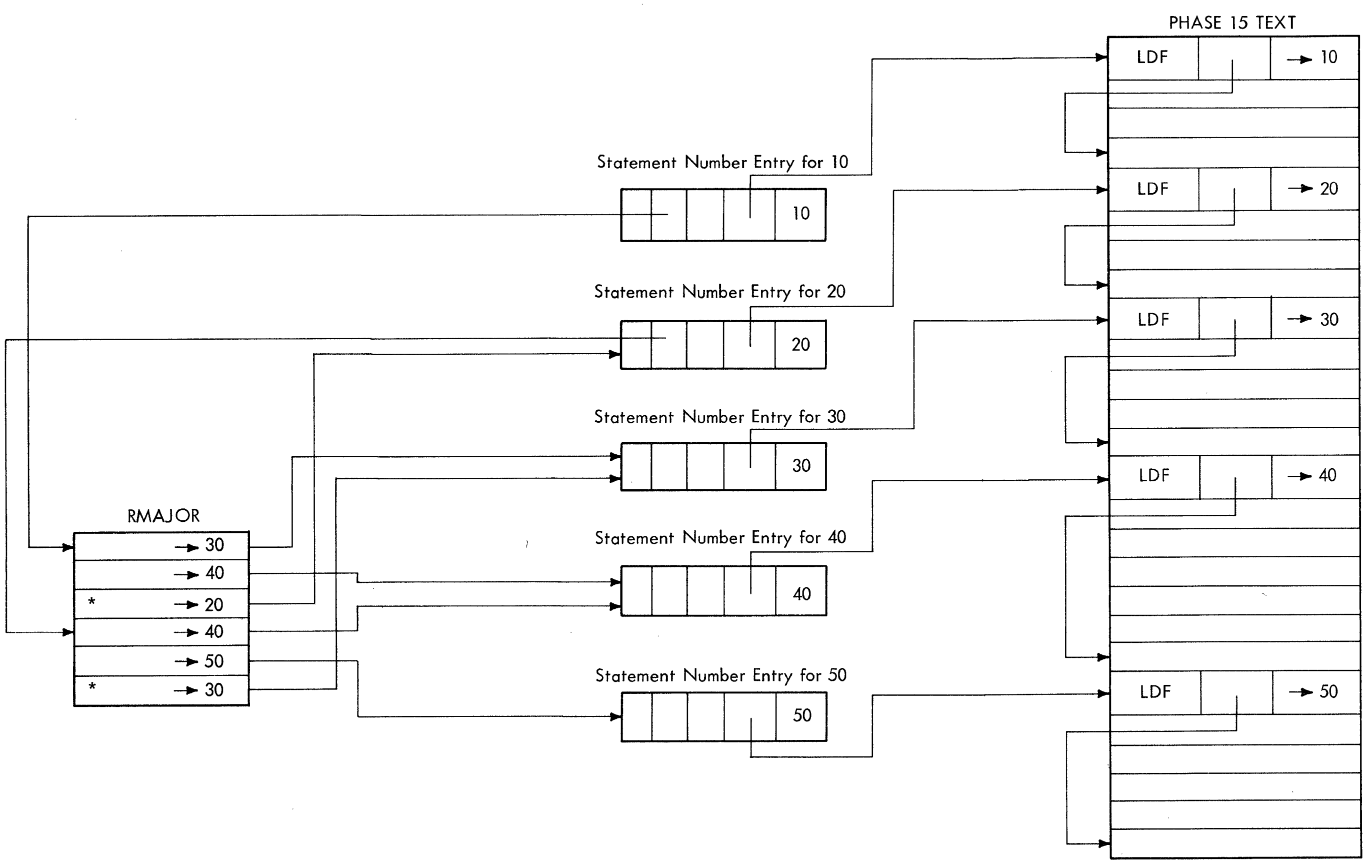


Figure 5. Forward Connection Information

into the appropriate field of the communication table (refer to 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 PHAZ15-IEKJA has performed this process for all pointers in the table, the entries in the statement number chain are arranged in the order in which their associated statement numbers are defined in the module. The new order of the chain also reflects the order 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 which text blocks receive control from which 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 C1520-IEKJA2. Each CMAJOR entry made by 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 (refer to "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; it then gathers the backward connection information for the block associated with the second entry in the chain; etc.

For each text block, 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 (refer to Appendix A, "Statement Number/Array Table"). The statement number entry thereby 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), PHAZ15-IEKJA obtains, in turn, each entry in the statement number chain. (The entries are obtained in the order in which they are chained.) After PHAZ15-IEKJA 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. (Recall that the RMAJOR entries for a block indicate the blocks to which that block may pass control.) 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 PHAZ15-IEKJA records a pointer to its associated statement number entry in the next available entry in CMAJOR. 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. PHAZ15-IEKJA flags the last entry in CMAJOR for block I. When the statement number chain has been completely searched, PHAZ15-IEKJA has gathered all the backward connection information for block I. Each entry that PHAZ15-IEKJA 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 above 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 figure, 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.

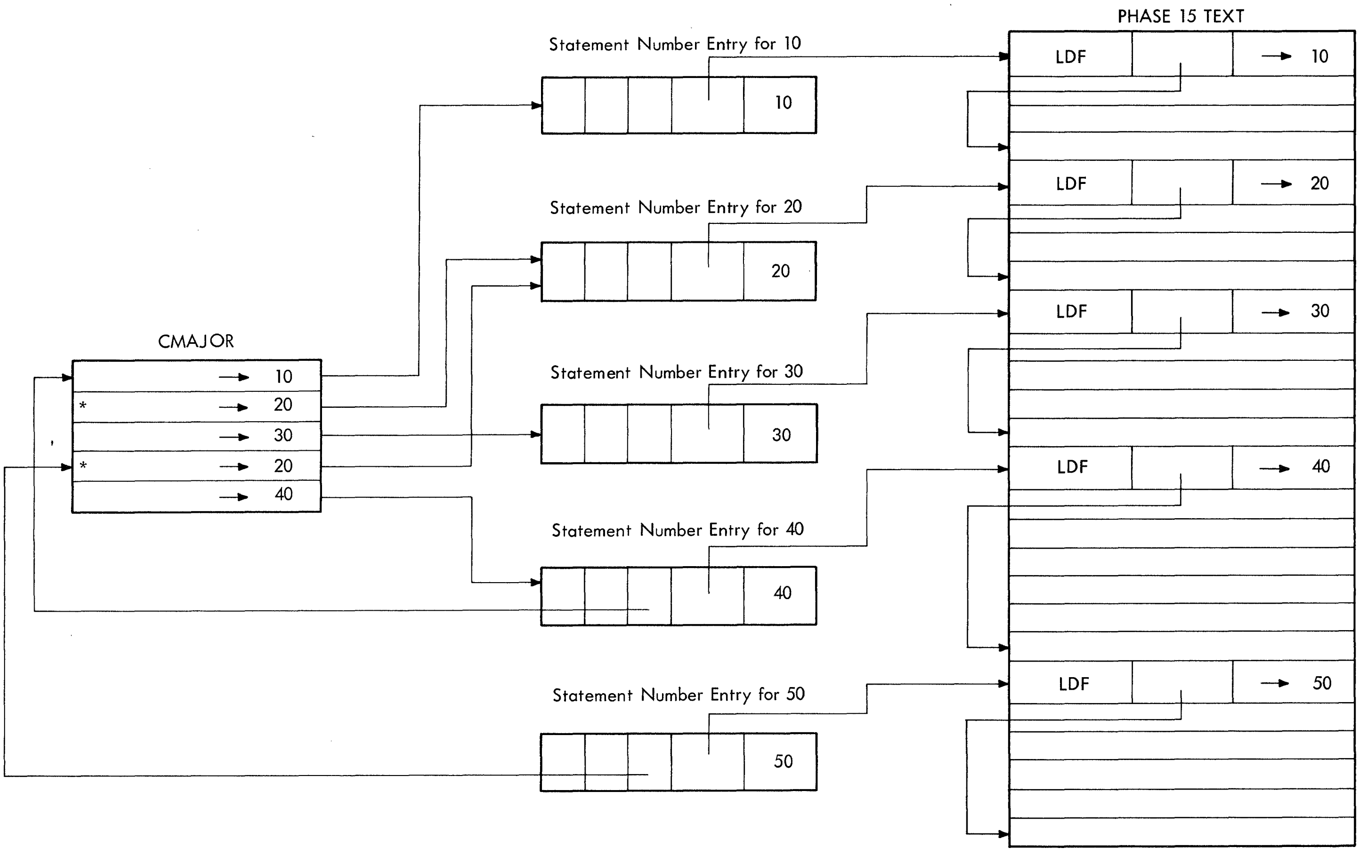


Figure 6. Backward Connection Information

- 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.

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 CORAL-IEKGCR are the routines which process constants, local variables, and external references.

CORAL-IEKGCR 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 order 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 NDATA-IEKGDA 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 two initial housekeeping entries):

Adjective Code for:	Pointer
	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 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. The reader should refer to Appendix B, "Phase 15/20 Intermediate Text Modification" for the detailed format of the phase 15 data text entry and the use of the special fields not discussed.

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 initially set to zero and 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:

- Contain the displacement to be assigned to the next operand.

- Determine when the next address constant is to be established. (When the location counter achieves 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.

* Hollerith character strings when used as arguments.

* 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, variables, and Hollerith character strings are processed in the same manner, they are described together.

Constants, Variables, and Hollerith Character Strings Used as Arguments: Subroutine CORAL-IEKGCR first assigns relative addresses to the constants of the module. As each constant is assigned a relative address, 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, variables, and Hollerith character strings 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 object module are not explicitly assigned an address constant. Such operands use the base address value loaded into reserved register 12 as their address constant (refer to Phase 20, "Branching Optimization"). 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. The displacement is recorded in the information table entry for that operand. The location counter is then updated, and tested to see if it exceeds 4095. If it does not, the next operand is processed as described above.

If sufficient operands exist to cause the location counter 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 location counter 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. The displacement for these operands is equal to the value of the location counter at the time they are considered for relative address assignment.

When the location counter 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 location counter is then reset to zero and the next operand is processed. This overall process is repeated until all operands (constant, variables, and Hollerith strings) are processed. Source module arrays are then considered for relative address assignment.

Arrays: CORAL-IEKGCR then assigns 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 F.) 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.

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, CORAL-IEKGCR 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. 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 location counter. If the value of the location counter does not exceed 4095, CORAL-IEKGCR does not take any additional action before it processes the next array. If the location counter value exceeds 4095, CORAL-IEKGCR establishes a new address constant, resets the location counter, and processes the next array. After all arrays have relative addresses, 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. 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, 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 EQVAR-IEKGEV 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 offset of the element. The offset for an element is the number of bytes that the element is displaced from the head of the group (refer to "Common and Equivalence Processing"). EQVAR-IEKGEV then compares the computed relative address to the previously established address constants. If an address constant exists such that the difference between the computed relative address and the address constant is less than 4095, EQVAR-IEKGEV 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. EQVAR-IEKGEV then processes the next element of the group.

If the desired address constant does not exist, 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. EQVAR-IEKGEV then assigns the element a displacement of zero, and processes the next element of the group. When all elements of the group are processed, 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.

----

[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.

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, 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, EQVAR-IEKGEV 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 byproduct 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 offset computed for it during common and equivalence processing. From the offset of each array in the common block under consideration, EQVAR-IEKGEV subtracts the span of that array. The result then replaces the previously computed offset for the array. If the result of one or more of these computations yields a negative value, EQVAR-IEKGEV 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 relative address. This address consists of the negative address constant and a displacement equal to the absolute value of the address constant plus the offset of the element.

If the computations which subtract spans from offsets do not yield a negative value, 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 offset of the element.

If at any time the displacement to be assigned to an element exceeds 4095, EQVAR-IEKGEV 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 offset 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 MCOORD 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. (Refer to 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 order of appearance in the source program. IEKGCZ now chains the data text entries according to the order of relative addresses it assigns to variables. Thus data text entries are now chained in the same relative order in which the variables will appear in the object module. This order simplifies the generation of text card images by phase 25.

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 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.

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 IHCFCOMH with the information required to implement READ/WRITE statements using namelists (refer to 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. NLIST-IEKTNL obtains the above information from the information table.

NLIST-IEKTNL 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 or values 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, CORAL-IEKGCR calls IEKTLOAD (via IEKGCZ) to place an adcon in the object module for special references. 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-B usage field of each information table entry informs CORAL-IEKGCR if a particular reference belongs to one of these categories. For each special reference that CORAL-IEKGCR detects, IEKGCZ calls 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 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.

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 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 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 (refer to 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 (refer to 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 (refer to Chart 10). Phase 20 consists of a control routine (LPSEL-IEKPLS) and six routine 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 only executed 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 only used 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 <u>status</u>, 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 (refer to Appendix B, "Phase 20 Intermediate Text Modification"). The status information for an operand or base address indicates such things as whether or not 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 following 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 it is to generate code that retains operands and base addresses in registers, and whether operand 1 is to be stored.
2. Phase 20 makes note of 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.

Table 3. Item Types and Registers Assigned in Basic Register Assignment

Register	Item Type
Floating-Point Register	
0	Arithmetic text entry operands that are real.
2	Imaginary part of the result of a complex function.
General Purpose Register	
0-1	Arithmetic text entry operands that are integer, or logical operands.
5	Branch addresses and selected logical operands
6	Operands that represent index values.
7	Base addresses
14	1. Used for computed GO TO operations. 2. Logical result of comparison operations.
15	Used for computed GO TO operations.

Basic register assignment essentially treats System/360 as if 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.

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).
- 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 above 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 Figure 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 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, 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, SSTAT-IEKRSS 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. SSTAT-IEKRSS uses a similar procedure to set the status of operand 3.

To set the status of the base address of operand 2, SSTAT-IEKRSS determines the relationship of that base address to the current base address (see note). If they correspond, SSTAT-IEKRSS 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.

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.

SSTAT-IEKRSS 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 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, 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, SPLRA-IEKRSL assigns operand 2 a register. It selects the register according to the type of operand (refer to 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, 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 according to 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, SPLRA-IEKRSL does not assign a register to the base address of operand 2. SPLRA-IEKRSL 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, SPLRA-IEKRSL assigns the appropriate register (refer to Table 3) to that operand, and enters the number of that register into the R3 field.

Operand 1 is always assigned a register. SPLRA-IEKRSL selects the register according to the type of operand 1 (refer to 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 that it is to be stored into. If such is the case, 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 it is not to be stored into, 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 the control routine of phase 20, 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 the FSD.

Full Register Assignment - OPT=1

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 or 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-IEKRSX) passes control to REGAS-IEKRRG which 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 (MCOORD 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 outlined 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 REGAS-IEKRRG 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. 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.

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

4. FWDPAS-IEKRFP 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 REGAS-IEKRRG.

5. REGAS-IEKRRG passes control to the global assignment routine GLOBAS-IEKRGB, which performs global assignment for the module.

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 (LPSEL-IEKPLS).

Table Building for Register Assignment: 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 (refer to Appendix A, "Register Assignment Tables"). The local assignment tables can accommodate information for 100 text entries. PHAZ15 attempts to limit blocks to less than 100 text items. 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: 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) if 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, BKPAS-IEKRBP assigns a register to it. The register assigned is determined by consulting the register usage table (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.

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, BKPAS-IEKRBP 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, 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, BKPAS-IEKRBP 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, common variables and real operands cannot be assigned to registers across that reference. In addition, if the block contains a reference to a function subprogram, no local assignment may be made for real operands across the reference to that function. 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, 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: 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.

GLOBAS-IEKRGB 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.

A register usage table (RUSE) of the same type as described under local assignments (TRUSE) is used by the global assignment routine. For each category of operands, GLOBAS-IEKRGB 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. GLOBAS-IEKRGB 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, real and common variables are ineligible for global assignment. If the module contains any references to function subprograms no global assignment can be performed for real quantities. In other words, if a module contains both a reference to a subroutine and to a function subprogram, global assignment is restricted to integer and logical operands that are not in common.

Text Updating: 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, STXTR-IEKRSX interrogates the completed global assignment table to determine if a global assignment has been made for the operand. If it has, 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 STXTR-IEKRSX 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, 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 a register during global assignment, STXTR-IEKRSX 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. 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, 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 REGAS-IEKRRG upon completion of its functions. REGAS-IEKRRG, in turn, returns control to (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 that can be branched to by means of RX-format branch instructions. BLS-IEKSBS 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.

BLS-IEKSBS 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 encountered reference to an in-line routine.

Next BLS-IEKSBS computes all block sizes and determines those text blocks that can be branched to via RX-format branch instructions. A text block, once converted to machine code, can be branched to 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 BLS-IEKSBS to determine the source module blocks that can be branched to 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 15 counts the number of text entries that result from its processing and passes the information to phase 20.) For relatively small source modules (approximately 70 source statements), phase 20 reserves only one register. For sufficiently large source modules (approximately 280 source statements), a maximum of five 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 (refer to Fortran System Director, "Generation of Initialization Instructions") 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 adcon table (see Figure 11), certain text blocks are displaced less than 4096 bytes from an address in a reserved register. Such blocks can be branched to by RX-format branch instructions that use the address in a reserved register as the base address for the branch.

To determine the blocks that can be branched to via RX-format branch instructions, 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. If a block displaced less than 4096 bytes from that address exists, BLS-IEKSBS enters the displacement of that block (from the address) into the statement number entry. It also places in that statement number entry an indication that the block can be transferred to via an RX-format branch instruction, and records the number of the reserved register to be used in that branch instruction.

When BLS-IEKSBS has processed all blocks displaced less than 4096 bytes from the address in register 13, it processes those 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 order 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 outward. 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 first 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.

----

[1]The text optimization process relocates text entries from within a 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 7 illustrates the concept of back dominators. Each block in the figure 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.

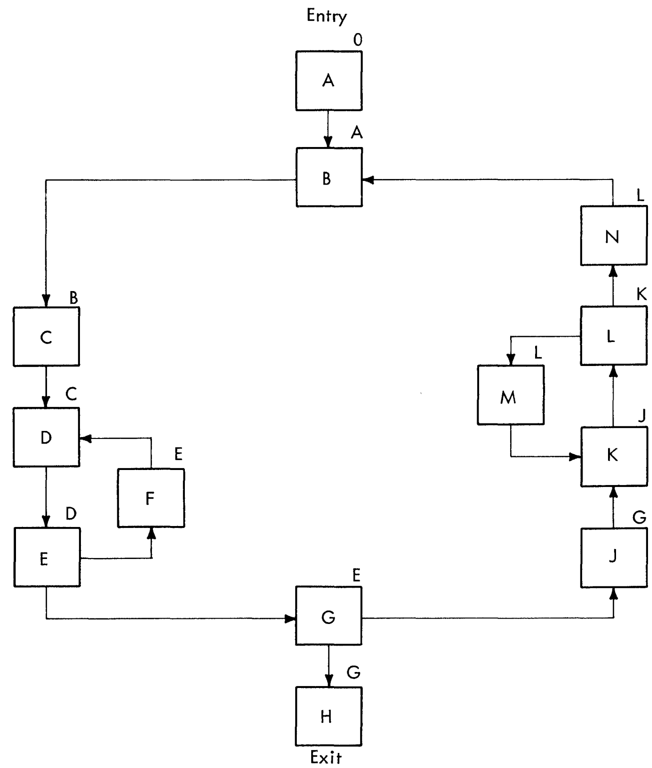


Figure 7. Back Dominators

When all back dominators are identified, a back target and a depth number for each text block are 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 order in which the loops are to be processed.

Figure 8 illustrates the concepts of back targets and depth numbers. Again each block in the figure represents a text block, which is identified by a single letter name. In this figure, 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.

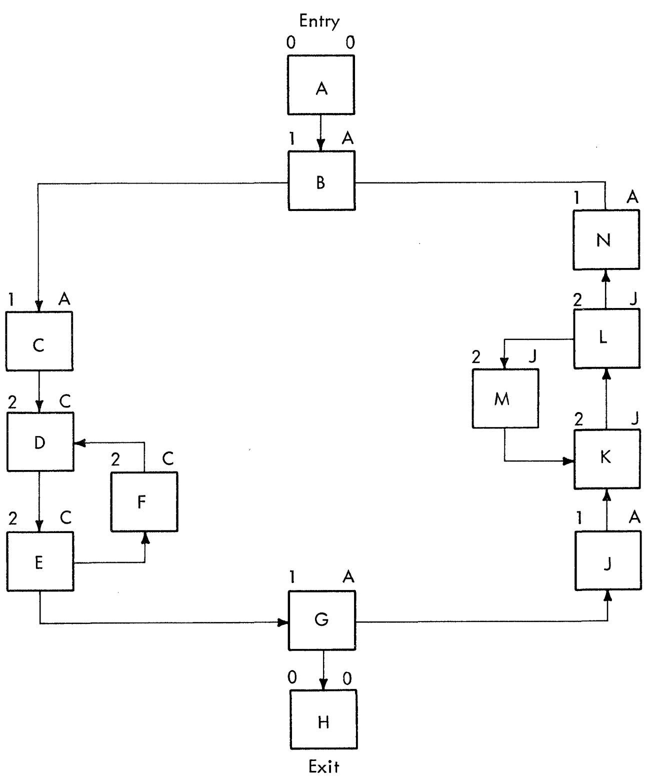


Figure 8. Back Targets and Depth Numbers

When the back target and depth number of each text block has been determined, loops are identified and the order in which they are to be processed is specified. The loops are ordered 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 order 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 order loops for processing.

<u>Determination of Back Dominators</u>

Subroutine TOPO-IEKPO determines the back dominator of each text block by examining the connection information for that block. The first block processed by 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.

TOPO-IEKPO assigns 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 (refer to Appendix A, "Statement Number/Array Table"). TOPO-IEKPO 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. 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, TOPO-IEKPO assigns it a provisional back dominator, which is the entry block of the module. All blocks on the first level are processed in this manner.

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. TOPO-IEKPO 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, TOPO-IEKPO assigns 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. 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. TOPO-IEKPO 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. Block J is the nearest block in the chain of back dominators of block I.

2. Block J has only one forward connection.

3. There exists a path from block I to itself that does not pass through block J.

If a block J exists that satisfies all the above conditions except the second, then the back target of block J is also the back target of block I.

If a block J satisfying conditions 1 and 3 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 order as back dominators are determined for them. The first block of the module is the first processed; first-level blocks are considered next; etc.

BAKT-IEKPB 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 (refer to Appendix A, "Statement Number/Array Table").

The processing performed by BAKT-IEKPB for each other block 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, 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, 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, BAKT-IEKPB assigns that block as the back target of the block under consideration. BAKT-IEKPB 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. 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, BAKT-IEKPB assigns its back target as the back target of the block under consideration. BAKT-IEKPB 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), BAKT-IEKPB obtains and examines the nearest block to block I in its chain of back dominators that has two or more forward connections. BAKT-IEKPB 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 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, 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 BAKT-IEKPB 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, 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. BAKT-IEKPB 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, BAKT-IEKPB retraces it to determine if 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; it 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, BAKT-IEKPB 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 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. BAKT-IEKPB 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. (BAKT-IEKPB records the value 1 in the loop number field of the statement number entry for each block in that loop.) BAKT-IEKPB 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.) 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, 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 order 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, information indicating which variables are busy-on-exit from which text blocks must be gathered. 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 which 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 one. 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 loads and stores 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 only defined 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 only used 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 constants and base variables are only 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 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 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 blocks discussed above have been appropriately marked for common variables and arguments, 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 off. The busy-on-exit table and CMAJOR are then used to set 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 LPSEL-IEKPLS.

To convert not busy-on-entry information to busy-on-entry information, 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, BIZX-IEKPZ sets the bit off and proceeds to the next block. This process is repeated until the last text block has been processed.

After BIZX-IEKPZ has set 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, BIZX-IEKPZ obtains (from CMAJOR) pointers to all blocks that are backward connections of that entry. Each backward connecting block is examined to determine whether or not it meets one of three criteria, which are:

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

When blocks in all backward connecting paths have satisfied one of the criteria, BIZX-IEKPZ 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, BIZX-IEKPZ passes control to LPSEL-IEKPLS, 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 (H) Programmer's Guide.)

To produce the listing, 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, SRPRIZ-IEKQAA then identifies the source statements that make up a particular loop and flags them on the listing by corresponding loop number. SPRRIZ-IEKQAA also uses the previously gathered back dominator information to compute listing indentations for the statements. The indentations show dominance relationships; that is, SRPSIZ-IEKQAA indents the statements that form a text block from the statements that form the back dominator of that block.

LOOP SELECTION

The loop selection routine of phase 20 (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 control routine of phase 20 (LPSEL-IEKPLS) sets this parameter to one after the process of structural determination is complete. The loop selection routine TARGET-IEKPT 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 control routine again calls TARGET-IEKPT, 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 TARGET-IEKPT 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 TARGET-IEKPT 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 control routine passes control to the routines that compute the size of each text block and then to the routine that computes the displacements required for branching optimization.)

The loop selection routine TARGET-IEKPT uses the value of the loop number parameter as a basis for selecting the loop to be processed. TARGET-IEKPT 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 (refer to 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).
- A pointer to the forward 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 loop selection routine TARGET-IEKPT 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 TARGET-IEKPT saves a pointer to it for use in the subsequent processing of the loop.

Pointer to Forward Target

The text optimization and full register assignment routines place both relocated and generated text entries into the forward target of the loop. The forward target of a loop (if it exists) is the single block to which the loop passes control after its execution is complete.

To locate the forward target (if any), the loop selection routine TARGET-IEKPT analyzes the backward connection information (in CMAJOR) for each block that is not in the selected loop. It marks all such blocks that receive control directly from a block in the selected loop as exit blocks. If only one exit block exists, that block is the forward target of the loop. (The forward target must not be entered from a block not in the loop.) TARGET-IEKPT saves a pointer to the forward target for use in the subsequent processing of the loop.

If the above condition is not met, the loop does not have a defined forward target.

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, and loop selection routine TARGET-IEKPT 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, TARGET-IEKPT 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 loop selection routine TARGET-IEKPT 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. TARGET-IEKPT 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. TARGET-IEKPT forms it during its search for the forward target of the loop. TARGET-IEKPT 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 only used in the block or used before it is defined is busy-on-exit from the loop. TARGET-IEKPT sets 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 re-computations 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.

The first two of the above 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, 4, 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. 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.

Common Expression Elimination - OPT=2

The object of common expression elimination, which is carried out by subroutine XPELIM-IEKQXM , is to eliminate 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, 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, logical, or subscript operator. Once a candidate is found, XPELIM-IEKQXM 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

• Table 4. Text Entry Types

Type	Definition	Processed by
Type 1	A text entry having an absolute constant[1] in either the operand 2 or operand 3 position.	Backward Movement
Type 2	A text entry having stored constants[2] in both the operand 2 and operand 3 positions.	Backward 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 4	A subscript text entry.	
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.

[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.

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, 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, XPELIM-IEKQXM substitutes that operand for operand 2 of the candidate and converts the operator to a store.
- If, on the other hand, operand 1 is redefined, 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, XPELIM-IEKQXM 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 possible participation in 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 control routine of phase 20, 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:

- Contain an arithmetic or logical operator.

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

When a candidate is found, 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, BACMOV-IEKQBM 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, 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 control routine of phase 20, 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: BACMOV-IEKQBM 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.

BACMOV-IEKQBM selects as candidates for elimination any simple store in which operand 1 is a non-subscripted variable. Pointers to the candidates are passed to SUBSUM-IEKQSM which determines if elimination is indeed possible according to the conditions illustrated in Table 5. At the same time, 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.

BACMOV-IEKQBM then deletes those candidates so marked by 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 Below in Block	Operand 2 Redefined Before Operand 1 Definition	Operand 1 Used Below Operand 2 Redefinition
1. No	No	No	X
2. No	No	Yes	No
3. No	Yes	No	X
4. No	Yes	Yes	No
5. Yes	Yes	No	X
6. Yes	Yes	Yes	No
X = condition cannot exist because of previous characteristics of operands.			

Elimination of Text Entry Expressions Involving Integer Constants: During the scan of a block for text entries to be moved to the back target, 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, BACMOV-IEKQBM 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≥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 manners in which these sections implement reduction are essentially the same.

Elimination of Multiplicative Text: To eliminate multiplicative text, REDUCE-IEKQSR examines the loop being processed to determine if it is a candidate for strength reduction. The loop is a candidate if:

- The loop contains an inert text entry (a type 3 text entry).

- 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] (a type 5 entry).

- 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, 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, REDUCE-IEKQSR:

   a. Calculates a new constant (K) equal to the product of the absolute constants.

   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 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 K.

   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 that is tested to determine if the loop is to be reexecuted. The branch constant is the constant to 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

---

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

text entry is a stored constant, REDUCE-IEKQSR performs similar processing to that described above. However, prior to generating the inert text entry, it generates two additional text entries and places them into the back target of the loop. The first text entry multiplies the two constants. Operand 1 of this text entry becomes the additive constant in the generated inert text entry. The second text entry multiplies operand 1 of the first generated text entry by the branch constant. 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 above 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, REDUCE-IEKQSR examines the loop being processed to determine if 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 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

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 register loads in the most strategic sections of the 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. (Refer to "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, GLOBAS-IEKRGB assigns it the first available register. If the item was globally assigned in a previously processed loop, the global assignment routine then determines whether the register assigned to the item in the previously processed loop is currently available. If that register is available, GLOBAS-IEKRGB 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. GLOBAS-IEKRGB therefore assigns the item an available register different from that assigned to it in the previously processed loop. GLOBAS-IEKRGB 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, GLOBAS-IEKRGB 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, GLOBAS-IEKRGB

----

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

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 load text item is not required. If any globally assigned item is defined within the loop and is also busy-on-exit from the loop, GLOBAS-IEKRGB 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.

GLOBAS-IEKRGB 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, BLS-IEKSBS is given control to calculate the size of each block. When the sizes of all blocks have been calculated, BLS-IEKSBS uses the block size information to determine the blocks that can be branched to 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 form. 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 appearing 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 SYSPRINT 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. Error messages produced during phase 25 (if any) are also recorded on the SYSPRINT data set.

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.

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 form 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. The address field of the TXT record thereby 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 which allocates space	Phase which uses space
	Entry Code	STALL-IEKGST phase 10	STALL-IEKGST and phase 25
	Format Text and Literal Constants	STALL-IEKGST phase 10	STALL-IEKGST phase 10
Register 13 → (start of 4096 bytes)	Adcon for Register 13 / Epilogue Save Area / Adcon for Register 12	STALL-IEKGST phase 10	phase 25
4096 bytes	Branch Tables	STALL-IEKGST phase 10	phase 25
	Parameter Lists	phase 15	phase 25
	Constants, Variables, Arrays	CORAL phase 15	CORAL phase 15
Register 12 → (if needed)	Adcons	CORAL phase 15	CORAL phase 15
	Namelist Dictionaries	CORAL phase 15	CORAL phase 15
	Phase 20 Temporaries	phase 20	phase 25
	'B' Block Labels	phase 25	phase 25
	Object Program Instructions	phase 25	phase 25
	Epilogue	phase 25	phase 25
	Prologue	phase 25	phase 25
	Epilogue Secondary Entry Points	phase 25	phase 25
	Prologue Secondary Entry Points	phase 25	phase 25

•Figure 9. Storage Layout for Text Information Construction

Phase 25 constructs text information by:

- Reserving dictionary entries 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 for a subprogram and secondary entry points and entering these instructions into TXT records.
- Converting phase 15/20 standard text into System/360 machine code and entering the code into TXT records.

Chart 20 shows the logic of phase 25 processing, down to, but not including, conversion of text to machine code.

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 that is referred to 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, MAINGN-IEKTA only assigns a base and displacement for statement numbers that are associated with text blocks that can not be branched to via RX-format branch instructions.

After all statement numbers are processed, bases and displacements are likewise assigned for the statement numbers appearing in computed GO TO statements. MAINGN-IEKTA scans the branch table chain (refer to Appendix A, "Branch Table"), and assigns a base and displacement for each branch table entry. 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.

Main Program Entry Coding

To generate main program entry coding, phase 25 works with the FSD subroutine IEKTLOAD. After IEKTLOAD saves the contents of the general registers and loads reserved registers with their associated addresses, subroutine ENTRY-IEKTEN (phase 25) generates instructions that perform the following functions:

- Load register 15 with the address of IHCFCOMH.
- Branch and link to subroutine IBFINT (arithmetic interruption subroutine of IHCFCOMH) so that it can set the interruption mask.
- Load register 13 from register 4.
- Branch to apparent entry point.
- Load register 15 with the address of IHCFCOMH.
- Branch and link to STOP entry point in IHCFCOMH.
- Generate constant for STOP 0.
- Set up a save area that receives the contents of the main program registers, if a subprogram is called.
- Set up the address constants to be loaded into the reserved registers.

Note: At execution time, subroutine IBFINT is given control to set the interruption mask.

Text Conversion

Phase 25 converts intermediate text into Operating 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.
- I/O Statement Processing.
- CALL Statement Processing.
- Code Generation.
- RETURN Statement Processing.
- END Statement Processing.

The logic of text conversion is illustrated in Chart 21.

STATEMENT NUMBER PROCESSING: When the operator of the text entry indicates a statement number, MAINGN-IEKTA passes control to subroutine LABEL-IEKTLB. LABEL-IEKTLB 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 effected through the use of the relative address for that statement number.

Note: If branching optimization is being implemented, only statement number that can not be branched to via RX format branch instructions (i.e., statement numbers that are not within the range of registers 13, 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 if that statement number appears in a computed GO TO statement. If it does, 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 (refer to "END Statement Processing") when the text representation of the END statement is encountered.

I/O STATEMENT PROCESSING: When the operator of the text entry indicates an I/O statement, an I/O list item, or the end of an I/O list, MAINGN-IEKTA 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 I/O 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 I/O 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 requiring a FORMAT statement.
- The end of an I/O list associated with a READ/WRITE not requiring 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, 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. FNCALL-IEKVFN enters the calling sequence into TXT records.

CODE GENERATION: Code generation converts text entries having operators other than those for statement numbers and ENTRY, CALL, I/O, RETURN, and END 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 text entry 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 on in the appropriate positions of the bit strip such 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 if 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 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 on.

The bit strip is then matched against the skeleton array. Each skeleton instruction corresponding to a bit that is set 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 operand 1 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 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 (13) 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.)

<u>Branch Processing</u>: 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 if that instruction either:

- 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 if the instruction to be branched to 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 be branched to. 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 form D (0,Br) D represents the displacement of the instruction (to be branched to) from the address that is in the appropriate reserved register (Br).

If the instruction to be branched to 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 be branched to 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.

RETURN STATEMENT PROCESSING: When the operator of the text entry indicates a RETURN statement, MAINGN-IEKTA passes control to subroutine RETURN-IEKTRN, which generates a branch to the epilogue. The epilogue address is obtained from the subprogram 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 (refer to the section "Initialization Instructions").

END STATEMENT PROCESSING: When the operator of the text entry indicates an END statement, 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 ENTRY-IEKTEN to determine if the program being compiled is a main program or a subprogram and to take the appropriate action. If it is a subprogram, ENTRY-IEKTEN calls EPILOG-IEKTEP and PROLOG-IEKTPR. (Refer to "Prologue and Epilogue Generation.") If it is a main program, ENTRY-IEKTEN generates code to call IBFINT (arithmetic interruption subroutine of 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. ENTRY-IEKTEN returns control to END-IEKUEN.

END-IEKUEN places TXT and RLD records in the object module for the following: adcon for the save area, adcon for the Epilogue, adcon for register 12, if needed, adcons for branch tables, adcons for parameter lists, and adcons for 'B' block labels. END-IEKUEN generates TXT information for each temporary. END-IEKUEN calls IEND (FSD

----

[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.

entry point) to generate the loader END record which 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 entry point of the control section. END-IEKUEN then returns control to the FSD through 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

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 entry is a subroutine or function to the source program.
- 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 hex).

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 variables in that common block. A map of variable equivalenced into common is also provided.

In addition, 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.)

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 (on the SYSPRINT data set) appropriate messages for syntactical errors encountered during the processing of phases 10 and 15; its overall logic is illustrated in Chart 22. 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 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; the second 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.)

Message Processing

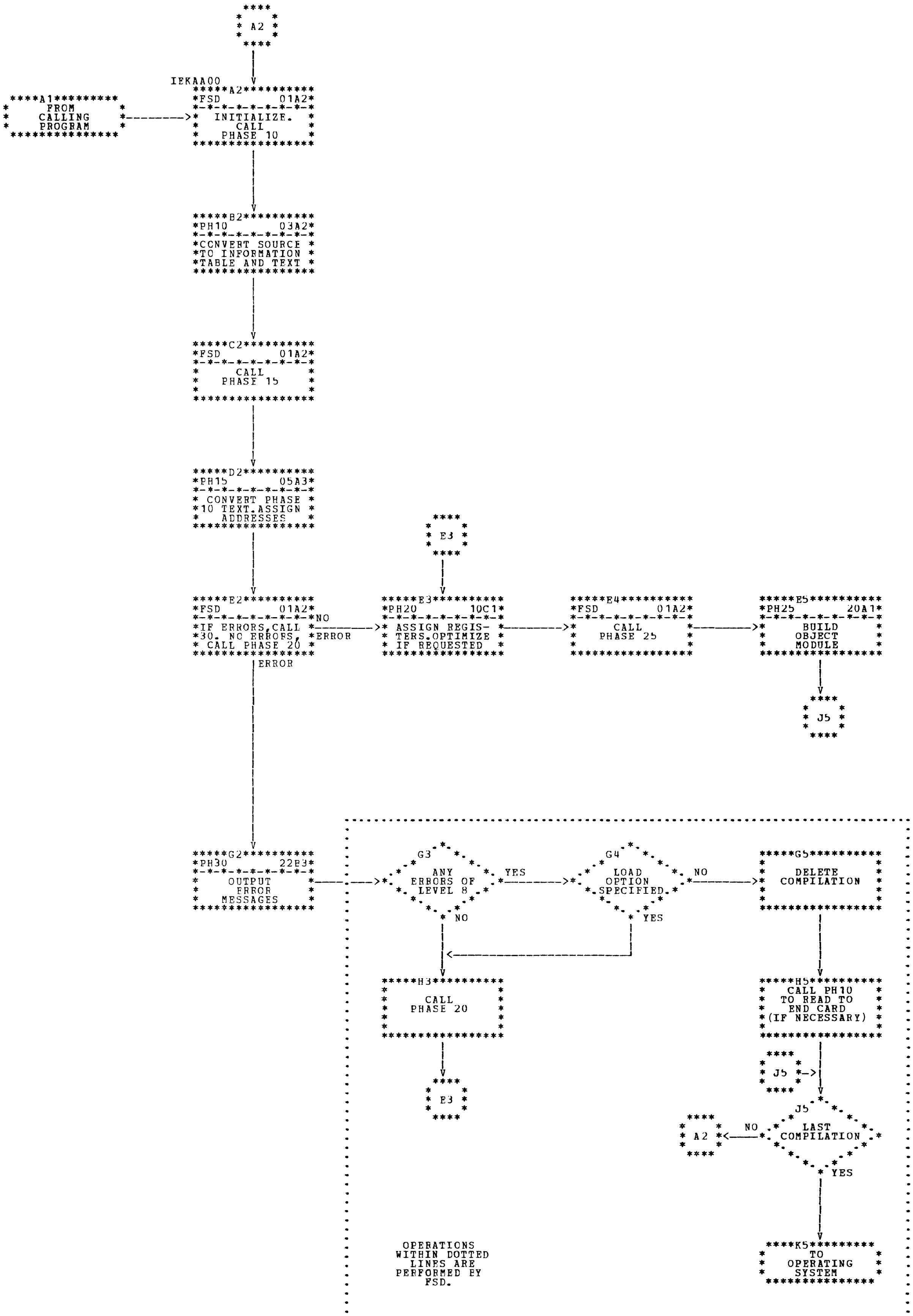
Using the message number in the error table entry multiplied by four, phase 30 locates, within the message pointer table (refer to Appendix A, "Diagnostic Message Tables"), the entry corresponding to the message number. This message pointer table entry contains (1) the length of the message associated 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.

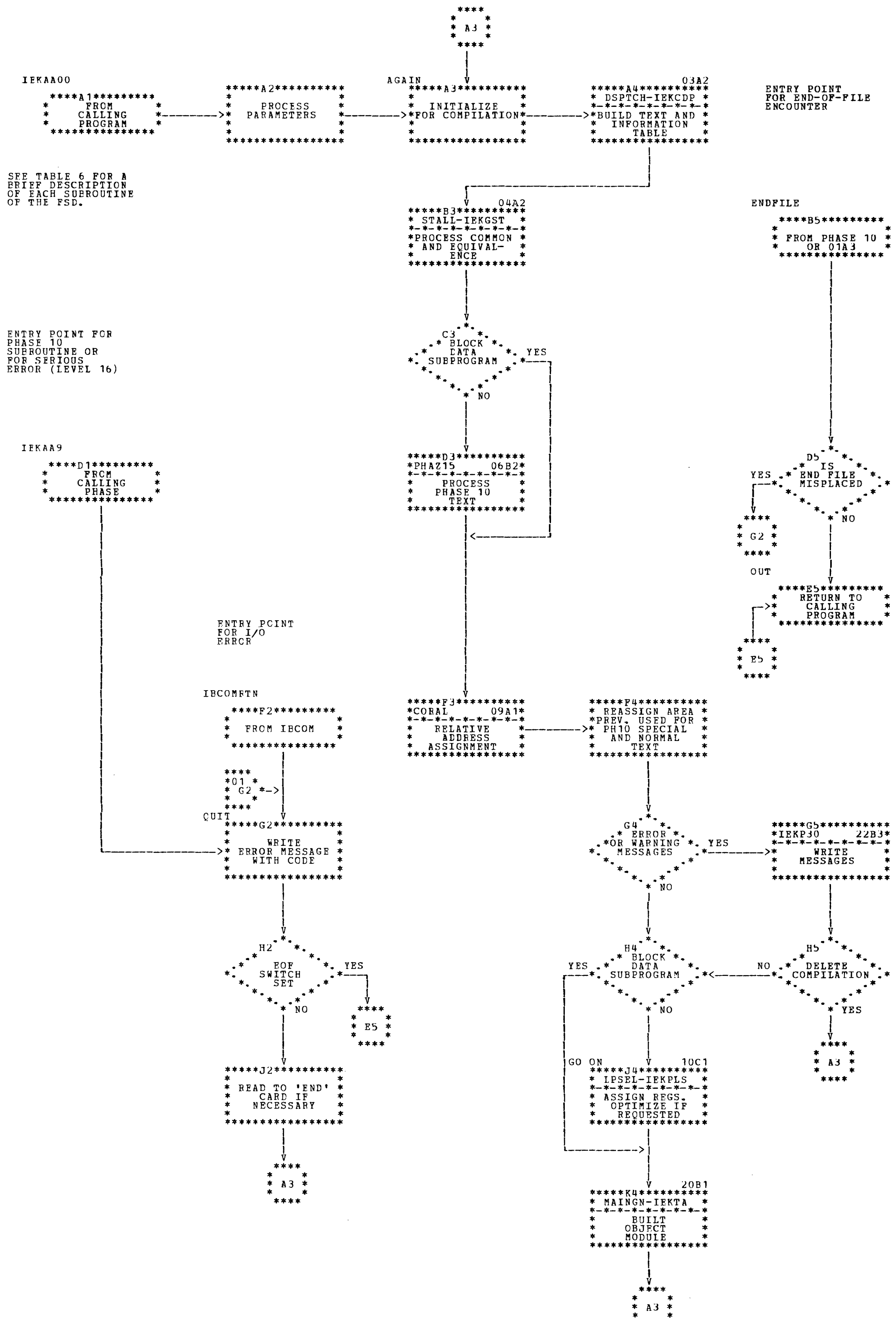
Having constructed the parameter list, phase 30 calls subroutine MSGWRT-IEKP31 which writes the message on the SYSPRINT data set. After the message is written, the next error table entry is obtained and processed as described above.

As each error table entry is being processed, the error level code (either 4 or 8) 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 encountered error. (See the publication IBM System/360 Operating System: FORTRAN IV (H) Programmer's Guide for explanations of all the messages the compiler generates.) 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 or 8) has been saved. The saved error level code is passed to the FSD when phase 30 processing is completed. This code is used by the FSD to determine whether or not the compilation is to be deleted.

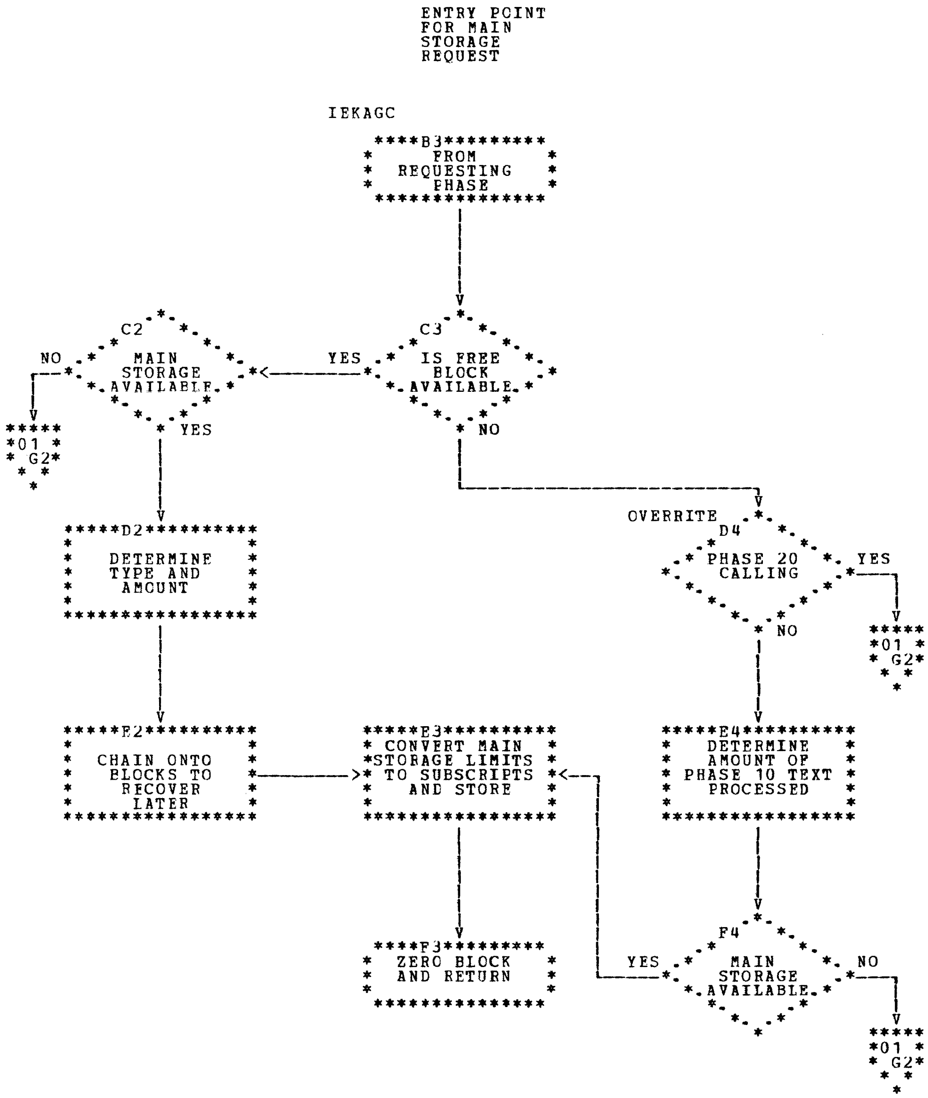
• Chart 00. Compiler Control Flow



• Chart 01. FSD Overall Logic



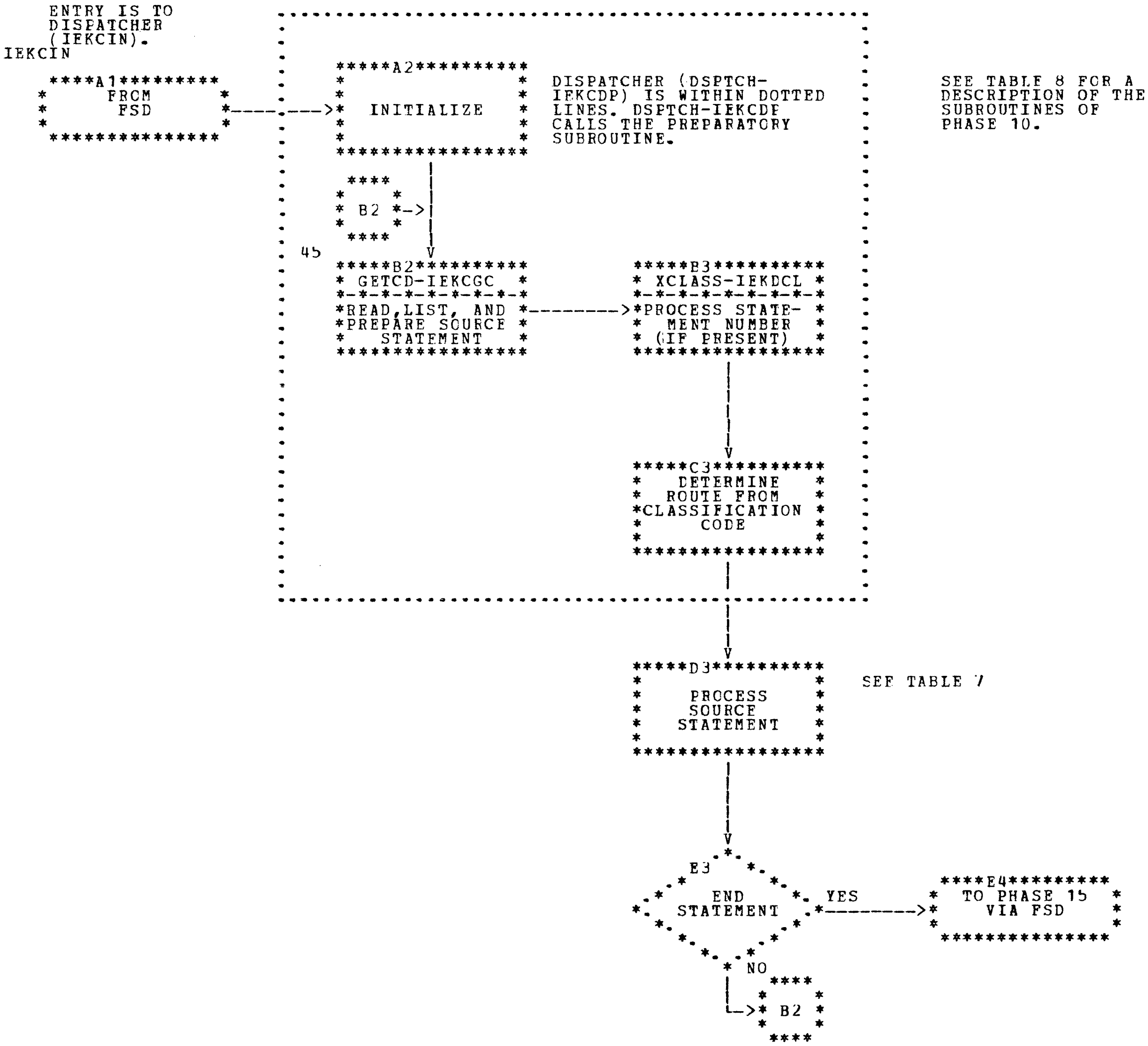
• Chart 02. FSD Storage Distribution



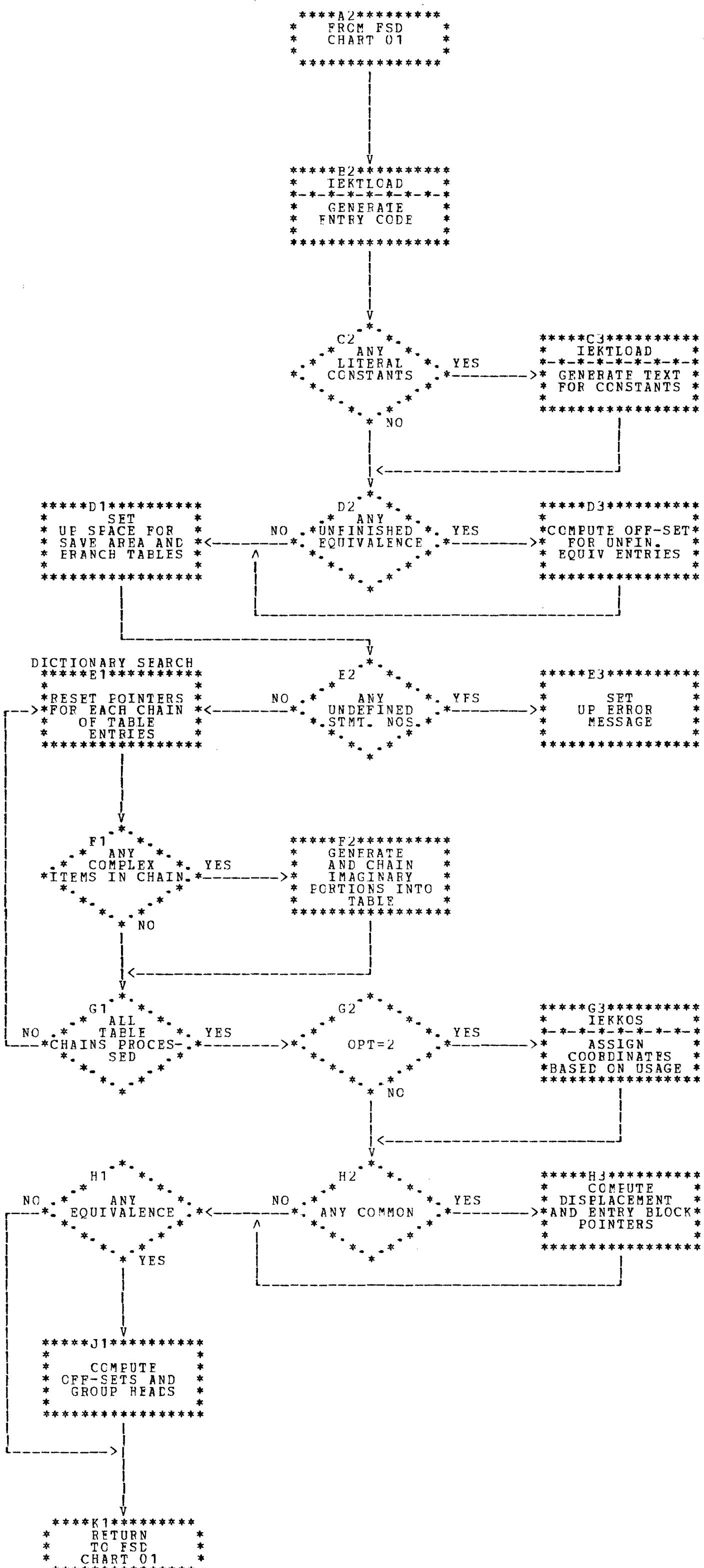
• Table 6. FSD Subroutine Directory

Subroutine	Function
AFIXPI-IEKAFP	Performs exponentiation of integers by integers.
IEKAAA	Communication table.
IEKAAD	Internal adcon table.
IEKAA00	Initializes compiler processing and calls the phases for execution. Entry point for compiler.
IEKAA01	Default options, & DDNAMES for compiler, PAGEHEAD.
IEKAA9	Deletes compilation if requested.
IEKAER	Error message table.
IEKAGC	Allocates and keeps track of main storage used in the construction of the information table and for collecting text entries.
IEKAPT	Maximizing service routine for integers and reals, diagnostic trace routine; bypasses IEKFCOMH for some error messages.
IEKATB	Provides diagnostic dumps of internal text and tables.
IEKATM	Timing routine.
IEKFCOMH	Controls formatted compile-time I/O. (Corresponds to IHCFCOMH; refer to Appendix E.)
IEKFIOCS	Interface between compiler, IEKFCOMH, and QSAM.
IEKTDC	Listing routine.
IEKTLOAD	Builds ESD, TXT, RLD, and loader END records.

• 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,EQUIV-ALENCE, 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 - IEKCIO	IEKCAR, IEKCCS, IEKCDP, IEKCGW, IEKCLT, IEKCPX, IEKCS1, IEKCS2, IEKCS3
DEFINE, 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

Subroutine	Type	Function
CSORN-IEKCCR	Utility (collection, conversion, entry placement)	Entry IEKCCR directs the entering of variables and constants into the information table. Entry IEKCLC converts integer, real, and complex constants to their binary equivalents. Entry IEKCS1 places variable names on full word boundaries for comparison to other variable names. Entry IEKCS2 places dictionary entries constructed for variables and constants of the source module into the information table. Entry IEKCS3 combines the functions of entries IEKCS1 and IEKCS2 (above) for variable names.
DSPTCH-IEKCDP	Dispatcher, Key Word, 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.
GETCD-IEKCGC	Preparatory	Reads, lists (if requested), packs, and classifies each source statement.
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.
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)	Phase 10 COMMON area.
PUTX-IEKCPX	Utility (entry placement)	Places text entries into the appropriate sub-blocks, obtains the next operator from the source statement, and places the operator in the text entry work area.

(Continued)

•Table 8. Phase 10 Subroutine Directory (Continued)

Subroutine	Type	Function
STALL-IEKGST	Utility (table entry and text generation)	Generates entry code for object module, translates 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	Key Word (table entry and text)	Develops intermediate text representations of DATA and TYPE statements, constructs information table entries for the operands of DATA and TYPE statements, and analyzes these statements for syntactical errors.
XDO-IEKCDO	Key Word (table entry and text)	Develops the intermediate text and information table entries for the DO statement and implied DOs appearing in I/O statements and analyzes them for syntactical errors.
XGO-IEKCGO	Key Word (table entry and text)	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	Key Word (table and text entry)	Develops intermediate text representations of I/O statements, constructs information table entries for their operands, and analyzes I/O statements for syntactical errors. (I/O list items are processed by subroutine XARITH-IEKCAR.)

(Continued)

• Table 8. Phase 10 Subroutine Directory (Continued)

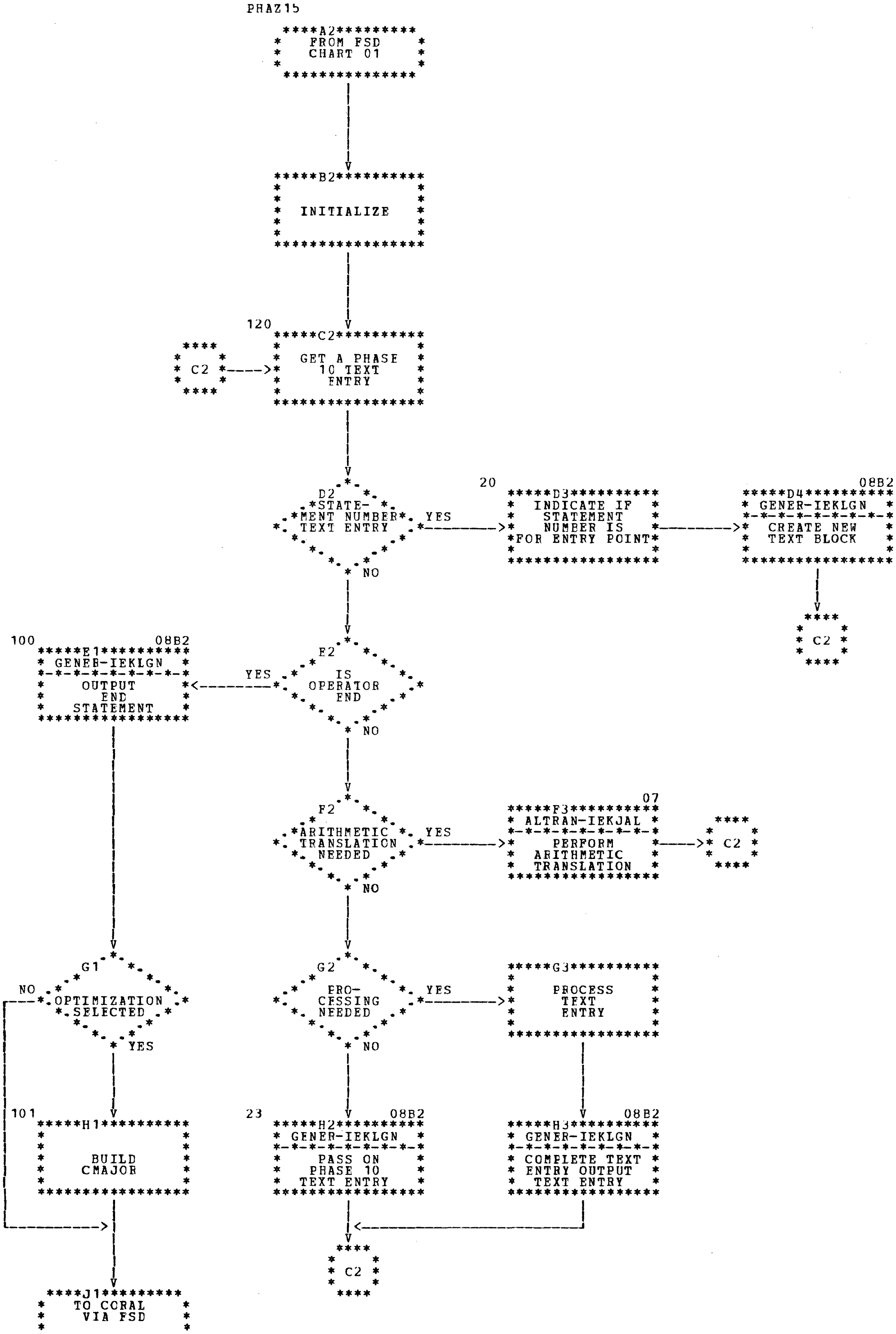
Subroutine	Type	Function
XSPECS-IEKCSP	Key Word (table entry)	Constructs information table entries for variables and arrays appearing in COMMON, DIMENSION, and EQUIVALENCE statements and analyzes these statements for syntactical errors.
XSUBPG-IEKCSR	Key Word (table and text entry)	Develops intermediate 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	Key Word (table entry and text)	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	Key Word (table entry and text)	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

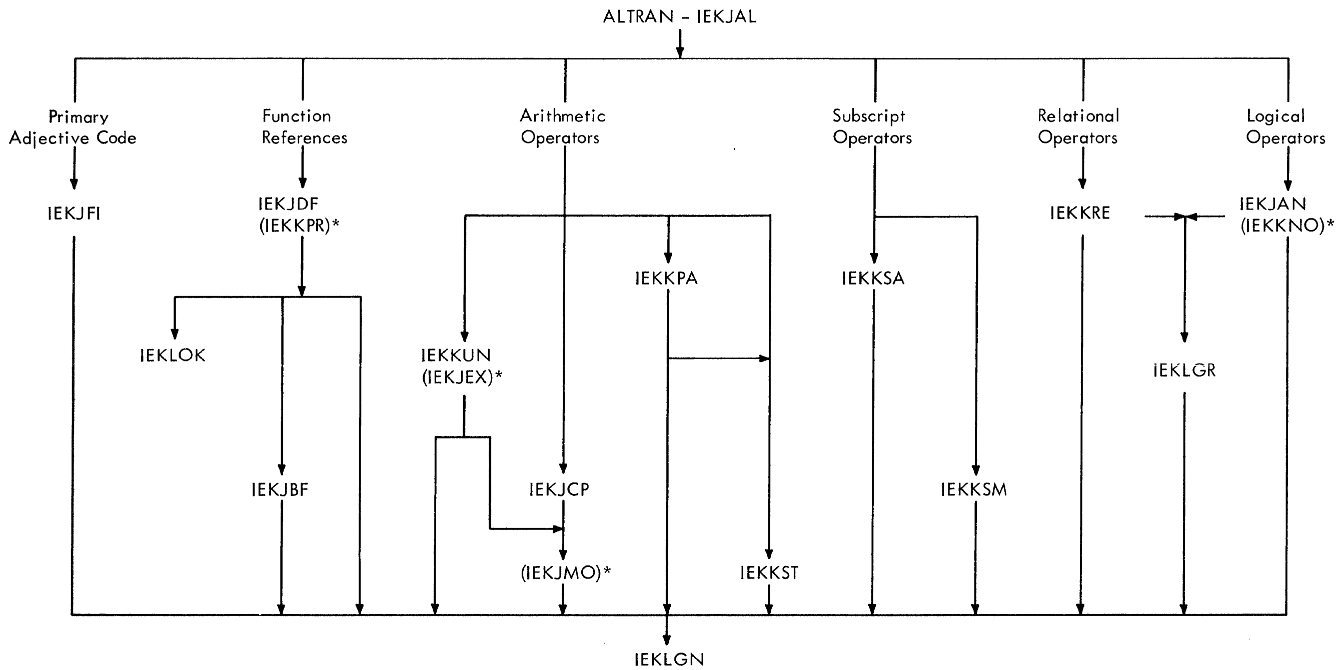
```
 ****A3*********
*   FROM FSD    *
*   CHART 00    *
*               *
 ***************
        |
        |
        |
        |
        |
        V
*****B3**********
*PHAZ15     06B2*
*-*-*-*-*-*-*-*-*
*    PROCESS    *
*   PHASE 10    *
*     TEXT      *
*****************
        |
        |
        |
        |
        V
*****C3**********
*CORAL      09A1*
*-*-*-*-*-*-*-*-*
*   RELATIVE    *
*    ADDRESS    *
*  ASSIGNMENT   *
*****************
        |
        |
        |
        |
        |
        V
 ****D3*********
*   TO PHASE    *
*  20 VIA FSD   *
*               *
 ***************
```

SEE TABLE 9 FOR A
BRIEF DESCRIPTION
OF THE SUBROUTINES
OF PHASE 15.

• 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.)

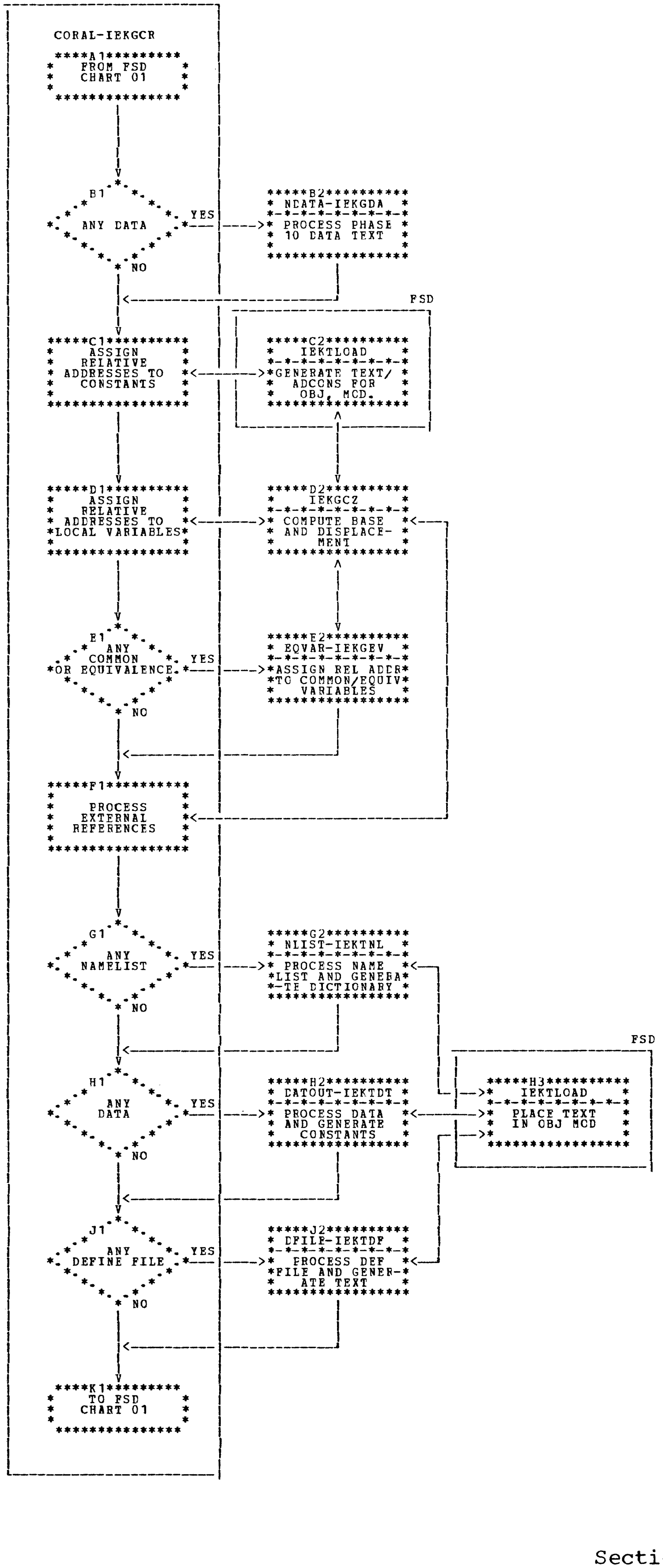
The specific functions of each of the subroutines associated with the arithmetic translator are given in the subroutine directory following the charts for phase 15.

• Chart 08. GENER-IEKLGN Text Generation

```
GENER-IEKLGN

    ****A2*********
   *     FROM      *
   *    CALLING    *
   *    ROUTINE    *
    ***************
           |
           V
   *****B2**********
   *               *
   *   INITIALIZE  *
   *               *
   *****************
           |
9004       V
   *****C2**********
   *  GET STORAGE  *
   *    FOR NEW    *
   *   TEXT ENTRY  *
   *****************                                                                  ****
           |                                                                         *    *
           |                                                                         * D5 *
           |                                                                         *    *
           V                                                                          ****
          .*.                                                                          |
    D2  .*   *.                199                         .                   9000     |
      .*  IS   *.           *****D3**********          *****D4**********               V
   .*  OPERATOR  *. NO      *               *          *   SET TEXT    *          ****D5*********
  *.  PHASE 15  .*--------->*    PASS ON    *--------->* CHAIN, BLOCK  *--------->*    RETURN     *
    *.  ITEM  .*            *   PHASE 10    *          *   SIZE, AND   *          *      TO       *
      *.    .*              *  TEXT ENTRY   *          *   BLOCK END   *          *    CALLER     *
        *..*                *****************          *****************           ***************
          * YES
          |
          V
          .*.
    E2  .*   *.                10
      .*       *.           *****E3**********                        TXTLAB-IEKLAB RECORDS
   .* STATEMENT  *. YES     * TXTLAB-IEKLAB *          ****          FALL-THROUGH CONNECTIONS
  *.   NUMBER   .*--------->*-*-*-*-*-*-*-*-*         *    *         AND SETS UP STATEMENT
    *.  TEXT  .*            *    RECORD     *-------->* D5 *         NUMBER TEXT ENTRIES.
      *.    .*              *  CONNECTION   *         *    *
        *..*                *  INFORMATION  *          ****
          * NO              *****************
          |
20        V
   *****F2**********        TXTREG-IEKLRG RECORDS CONNECTION
   * TXTREG-IEKLRG *        INFORMATION, OBTAINS DICTIONARY
   *-*-*-*-*-*-*-*-*        SPACE FOR TEMPORARIES AND UP-
   *    PROCESS    *        DATES MVS, MVF, AND MVX (VIA A CALL TO
   *    REGULAR    *        SUBROUTINE MATE - IEKLMA).
   *  TEXT ENTRY   *
   *****************
           |
           V
   *****G2**********
   *   SET TEXT    *
   * CHAIN, BLOCK  *
   *   SIZE, AND   *
   *   BLOCK END   *
   *****************
           |
           V
          ****
         *    *
         * D5 *
         *    *
          ****
```

• Chart 09. CORAL Overall Logic



•Table 9. Phase 15 Subroutine Directory

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)	Determines whether or not a given name represents a valid in-line function, and generates phase 15 text for that in-line function.
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.
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)	Processes data text.
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)	Outputs phase 15 text consisting of unchanged phase 10 text, phase 15 standard text, and phase 15 statement number text.
GENERTN-IEKJGR	PHAZ15 (5)	Builds appropriate phase 15 text entries for simple items forced from the pushdown table.
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, sets up error table entries for phase 30.

(Continued)

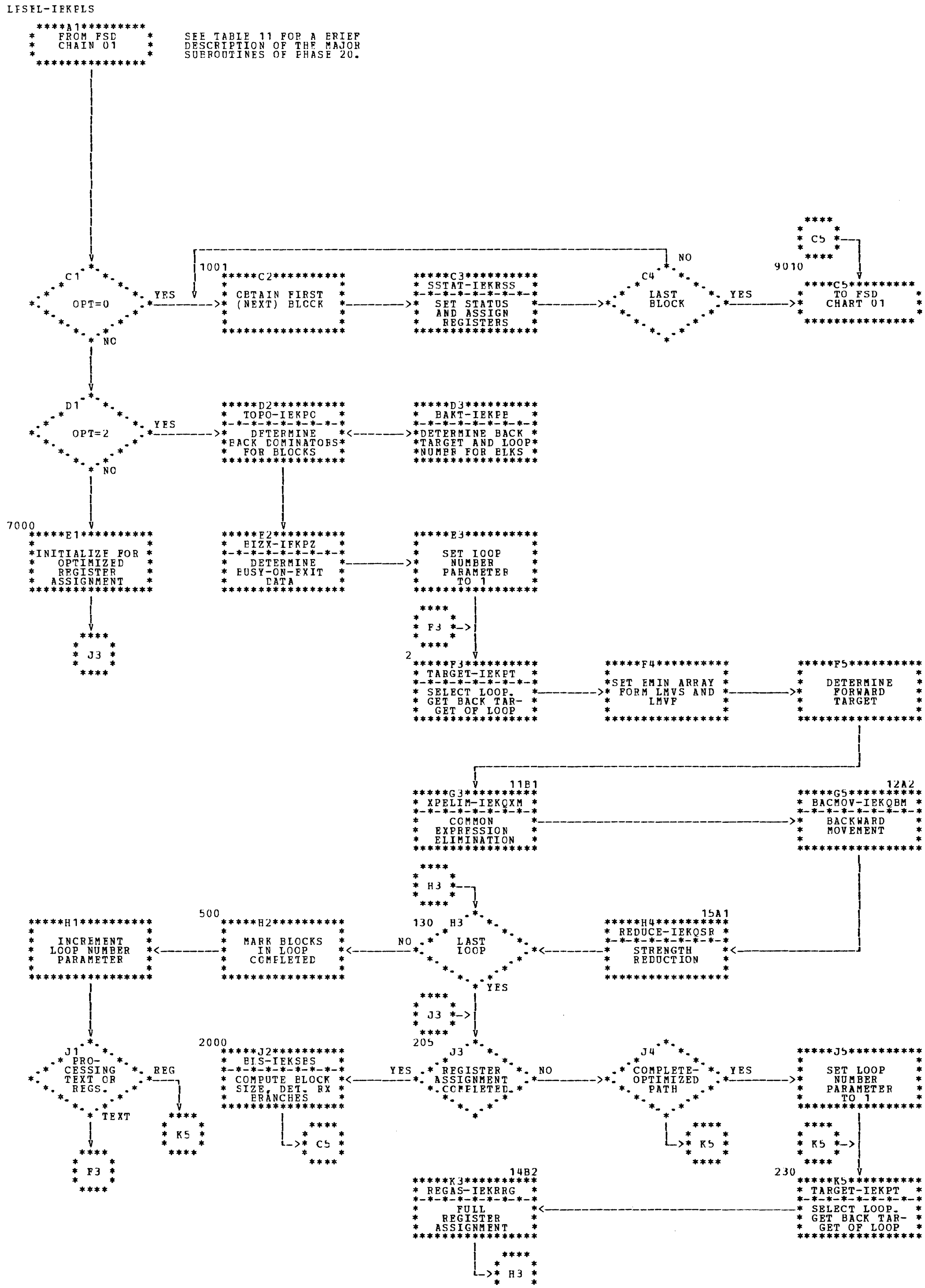
•Table 9. Phase 15 Subroutine Directory (Continued)

Subroutine	Associated Phase 15 Segment	Function
MATE-IEKLMA	PHAZ15 (5)	Records usage information in the MVS, MVF, and MVX fields if one of the optimized paths through phase 20 is selected.
NDATA-IEKGDA	CORAL (6)	Processes data text.
NLIST-IEKTNL	CORAL (6)	Processes namelist text.
OP1CHK-IEKKOP (IEKKNG) *	PHAZ15 (5)	Determines if 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.
RELOPS-IEKKRE	PHAZ15 (5)	Calls subroutine GENER-IEKLGN to output 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 output 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; creates entries in RMAJOR.
TXTREG-IEKLRG	PHAZ15 (5)	Processes standard phase 15 text entries for subroutine GENER-IEKLGN and makes RMAJOR entries.
UNARY-IEKRUN (IEKKSW) * (IEKJEX) *	PHAZ15 (5)	Optimizes arithmetic triplets and processes the exponentiation operator.
*Secondary entry points.		

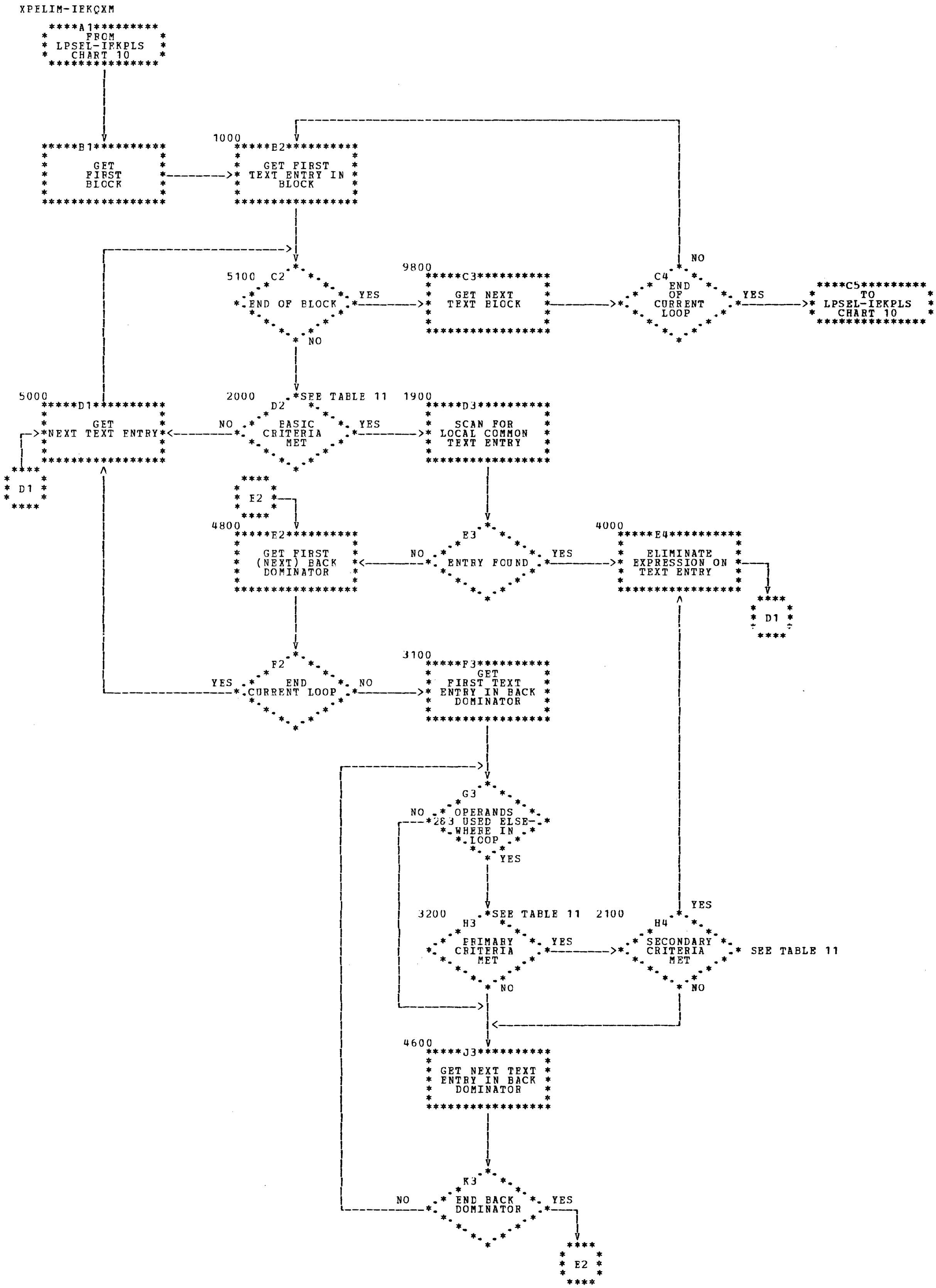
• Table 10. Phase 15 COMMON Areas

PH15-IEKJA1	Phase 15 common data area.
CMAJOR-IEKJA2	Backward connection table.
IEKJA3	Function information tables.
RMAJOR-IEKJA4	Forward connection table.
IEKLFT	Subprogram table.

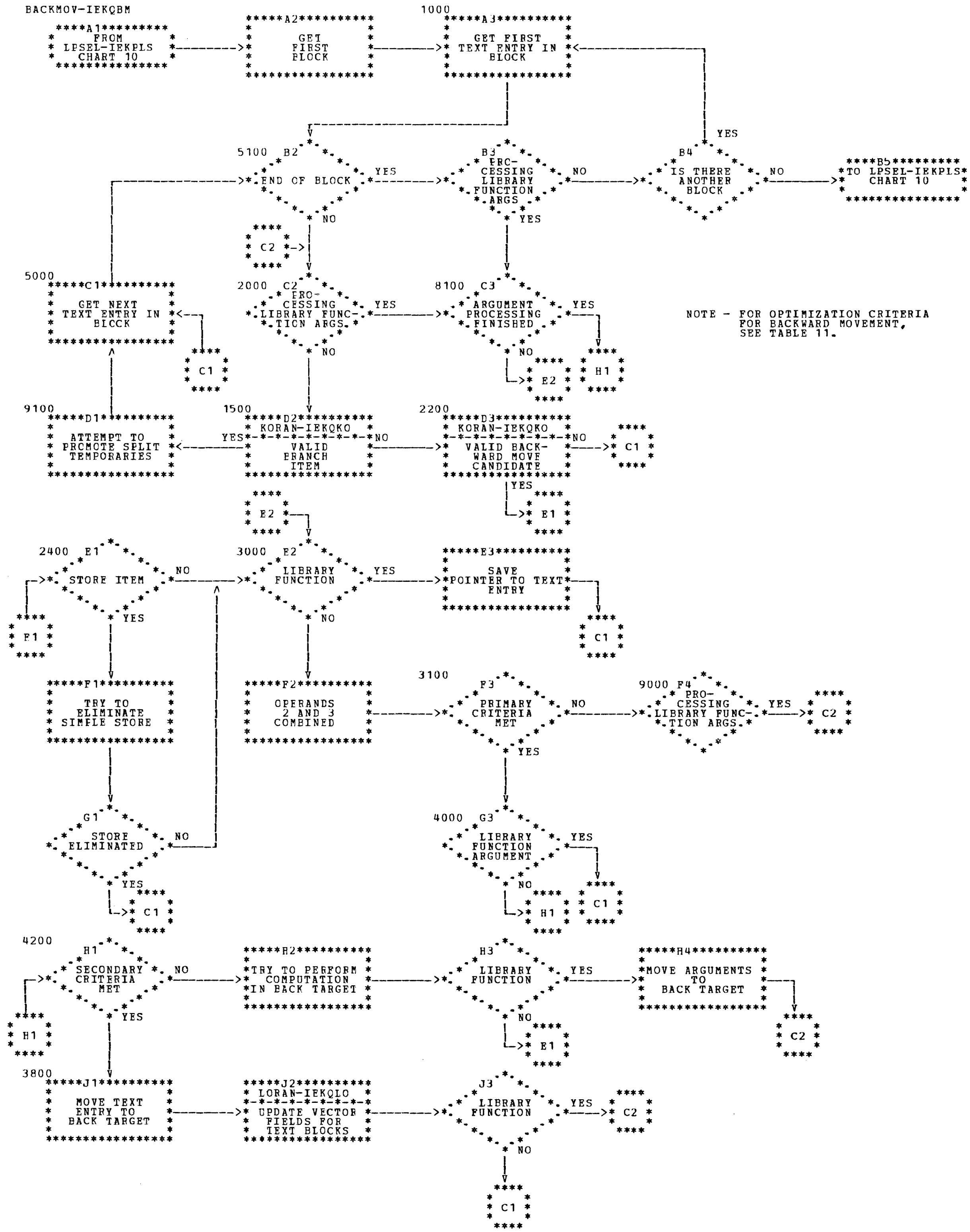
• Chart 10. Phase 20 Overall Logic



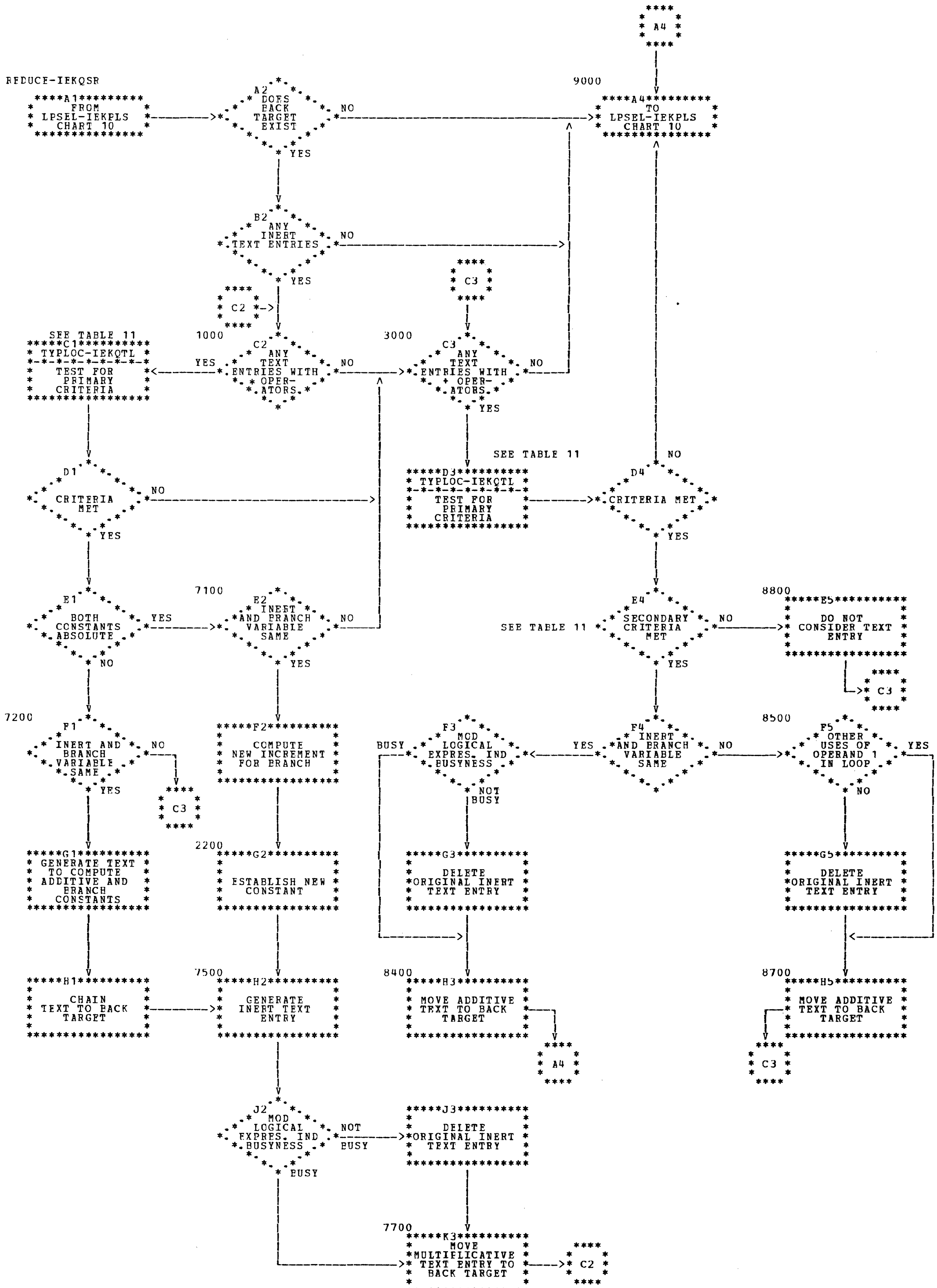
• Chart 11. Common Expression Elimination (XPELIM-IEKQXM)



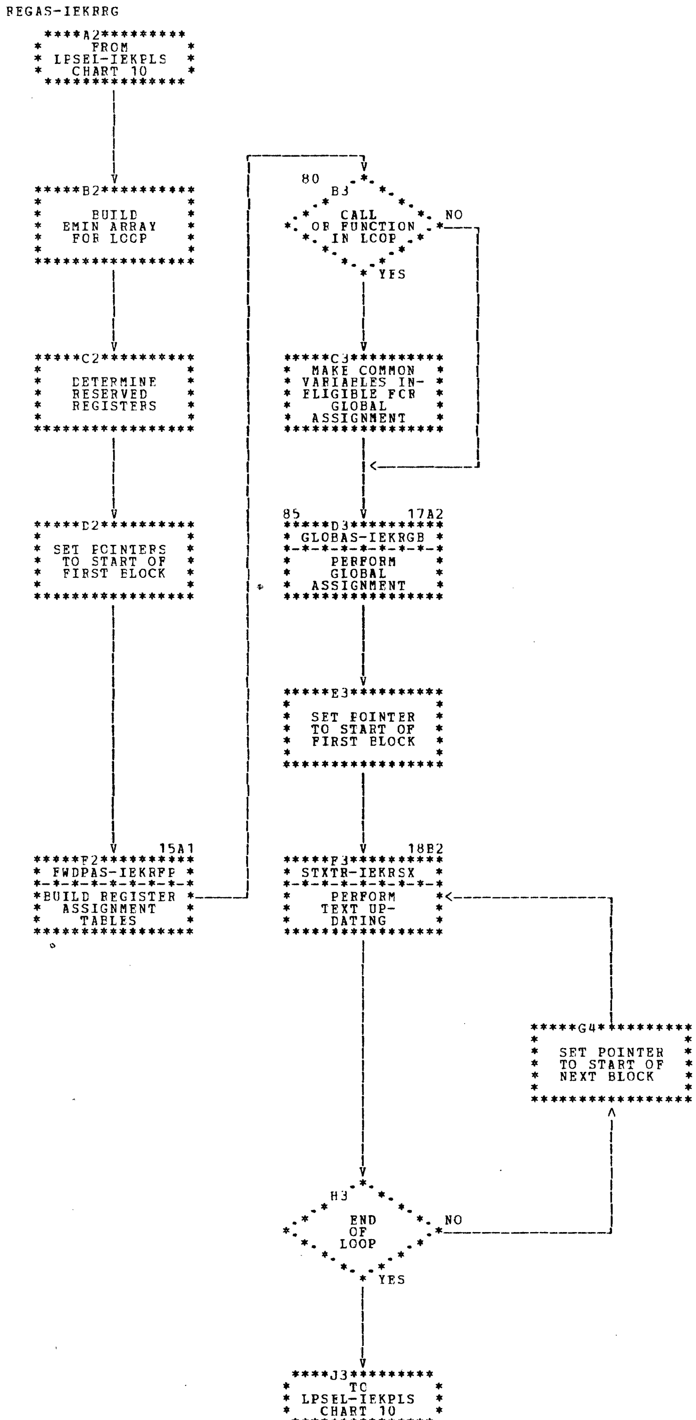
• Chart 12. Backward Movement (BACMOV-IEKQBM)



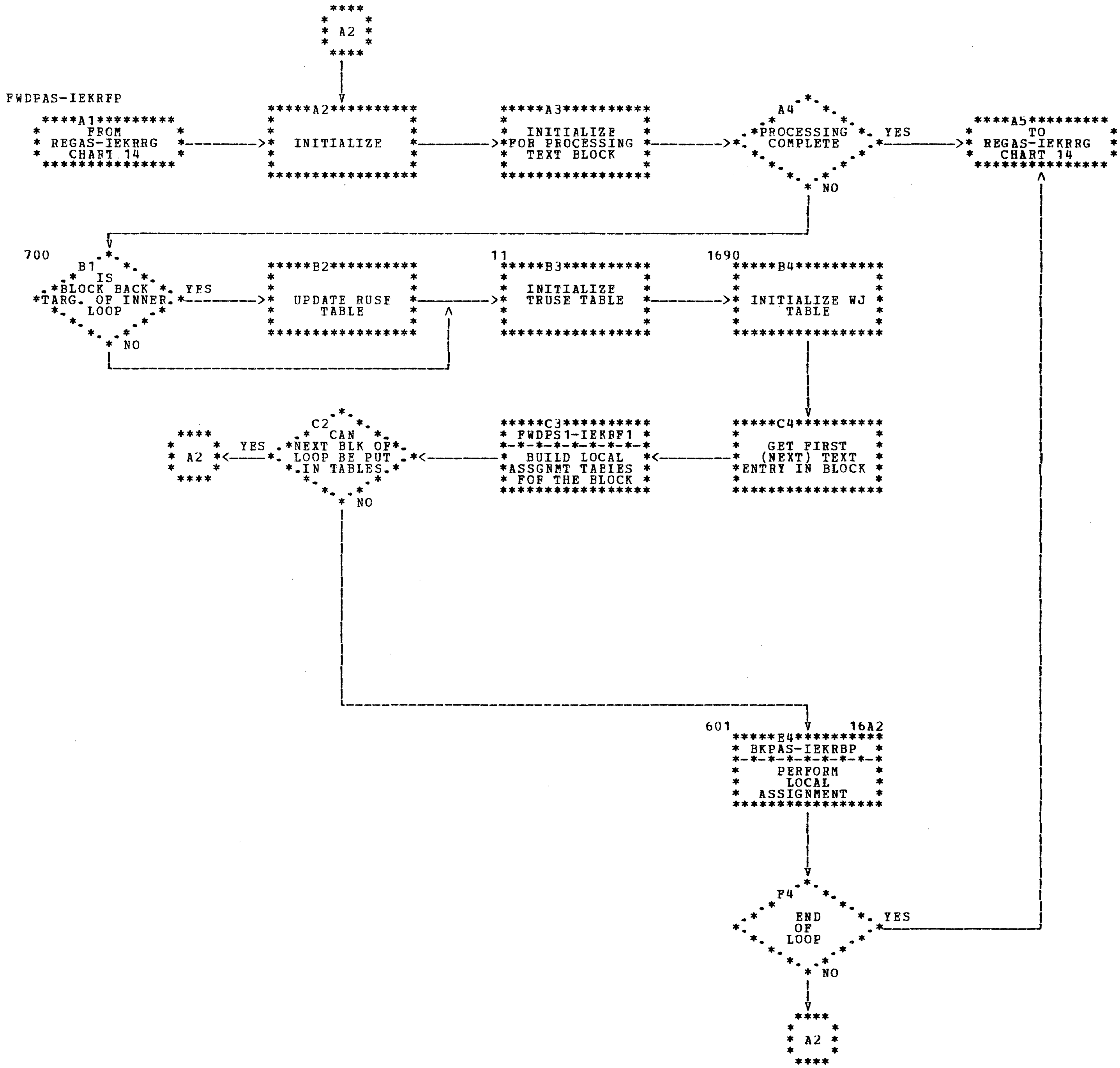
• Chart 13. Strength Reduction (REDUCE-IEKQSR)



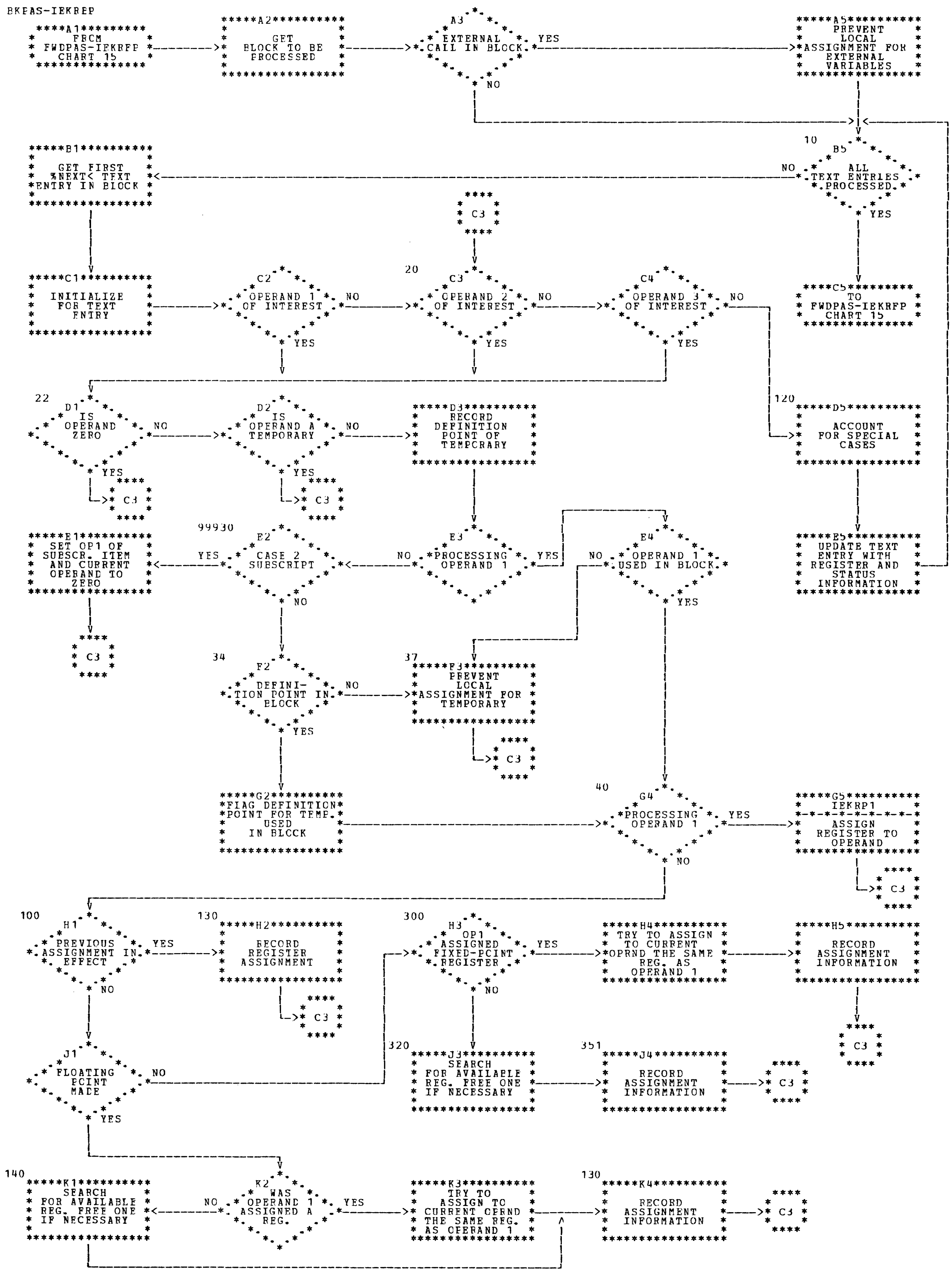
• Chart 14. Full Register Assignment (REGAS-IEKRRG)



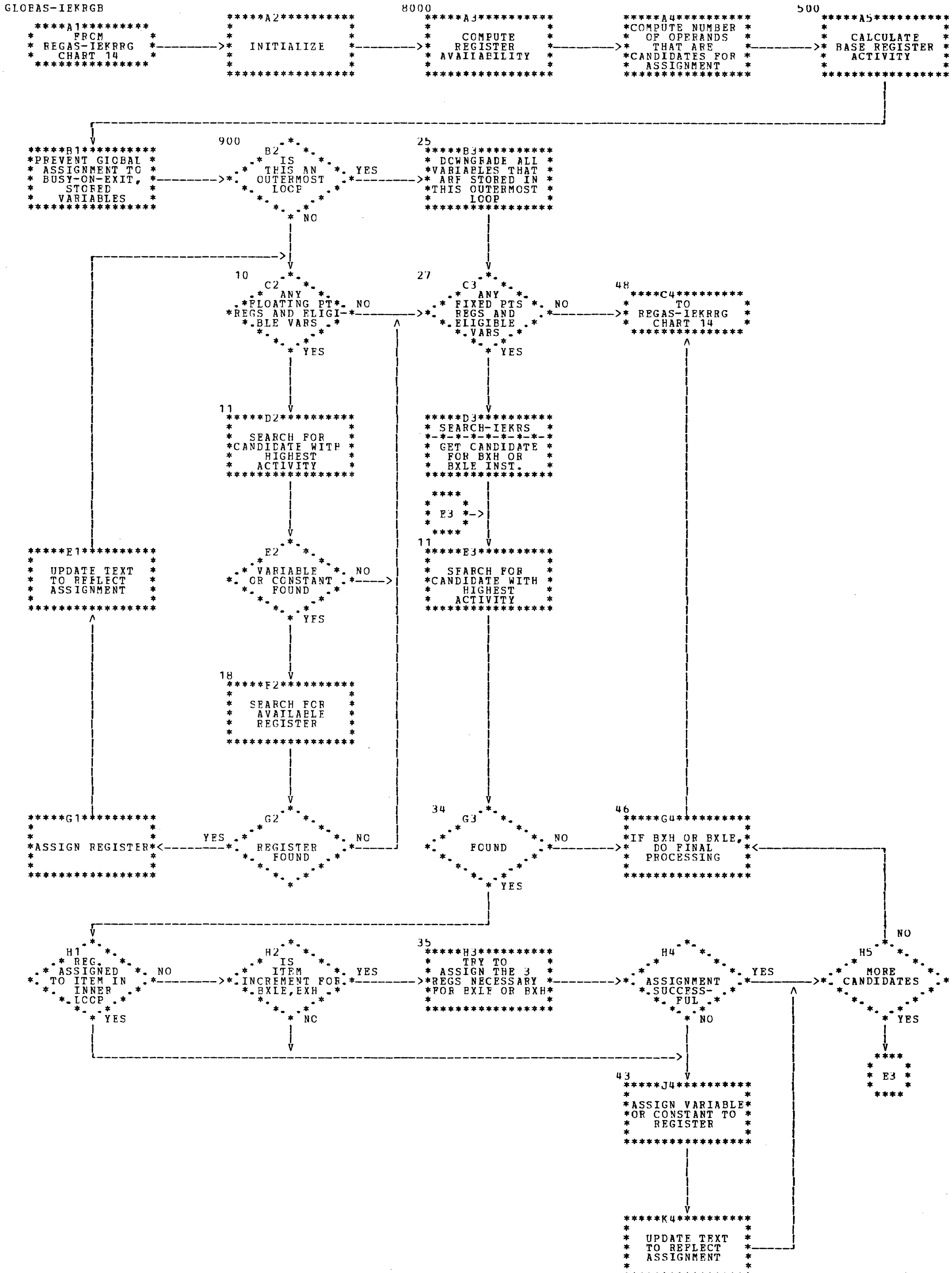
• Chart 15. Table Building (FWDPAS-IEKRFP)



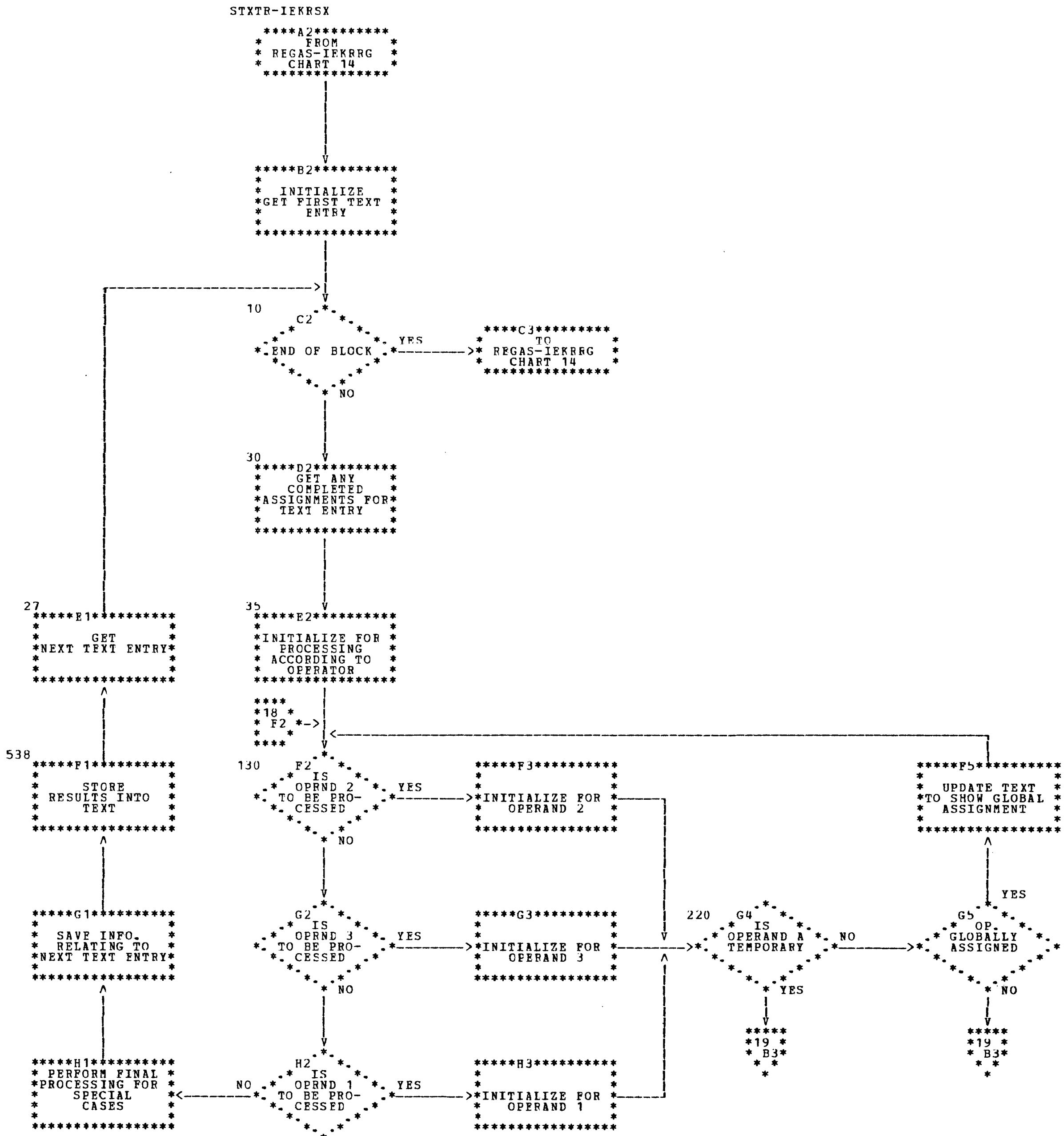
• Chart 16. Local Assignment (BKPAS-IEKRBP)



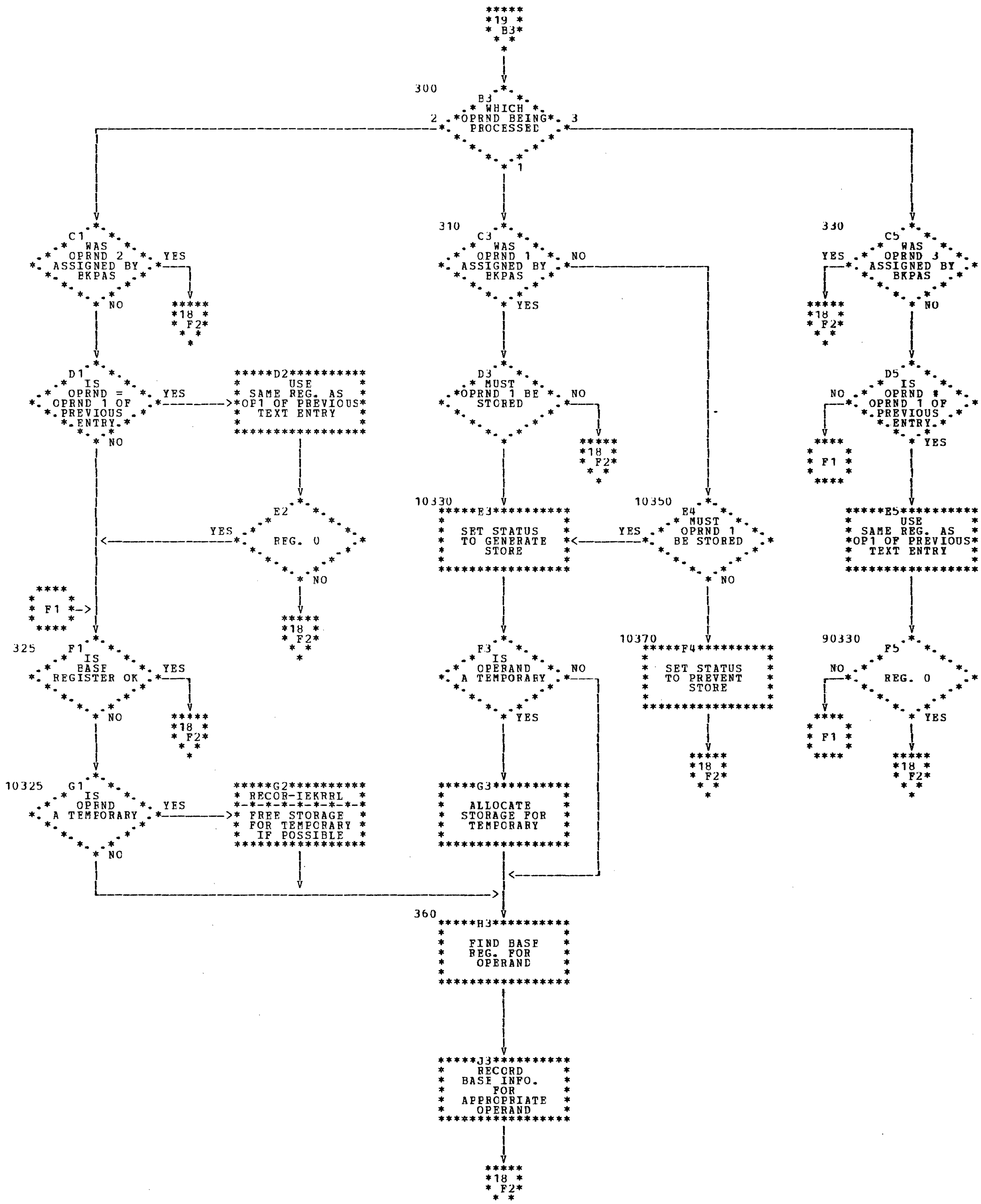
• Chart 17. Global Assignment (GLOBAS-IEKRGB)



• 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 or logical operator; 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

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	Printing routine for full register assignment.	Register assignment
BKPAS-IEKRBP	Control 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 branch instructions.	Branching optimization
CXIMAG-IEKRCI	Processes imaginary parts of complex functions during local register assignment.	Register assignment
FCLT50-IEKRFL	Performs special checks on text items whose function codes are less than 50.	
FOLLOW-IEKQF	Determines if intervening block causes redefinition of a variable.	Structural determination
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 if 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
INVERT-IEKPIV	Gets text pointers in a backward direction.	Text optimization
LOC-IEKRL1	Block data for register assignment.	
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
REGAS-IEKRRG	Controls full register assignment.	Register assignment
RELCOR-IEKRRL	Releases temporary main storage so it can be reused.	
SEARCH-IEKRS	Provides register loads upon entering the module.	
SPLRA-IEKRSL	Assigns registers during basic register assignment.	Register assignment

(Continued)

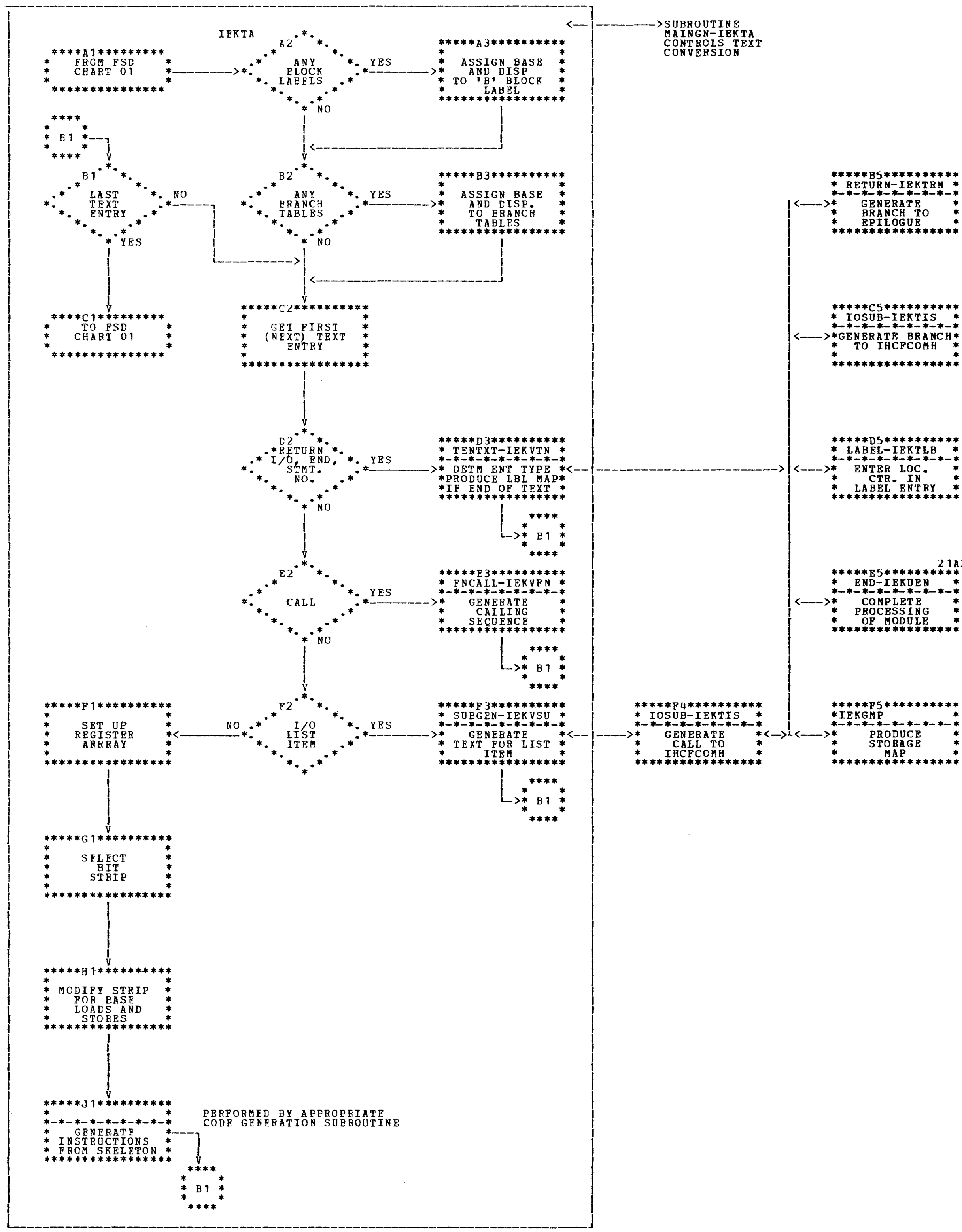
•Table 12. Phase 20 Subroutine Directory (Continued)

Subroutine	Function	Type
SSTAT-IEKRSS	Sets status information for operands and base addresses of text entries.	Text optimization
STXTR-IEKRSX	Controls text updating.	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
TNSFM-IEKRTF	Performs special checks on text items whose function codes are in the range of 50 to 55 inclusive.	
TYPLOC-IEKQTL	Locates interactions of text entries for strength reduction.	Text optimization
XPELIM-IEKQXM	Controls common expression elimination.	Text optimization

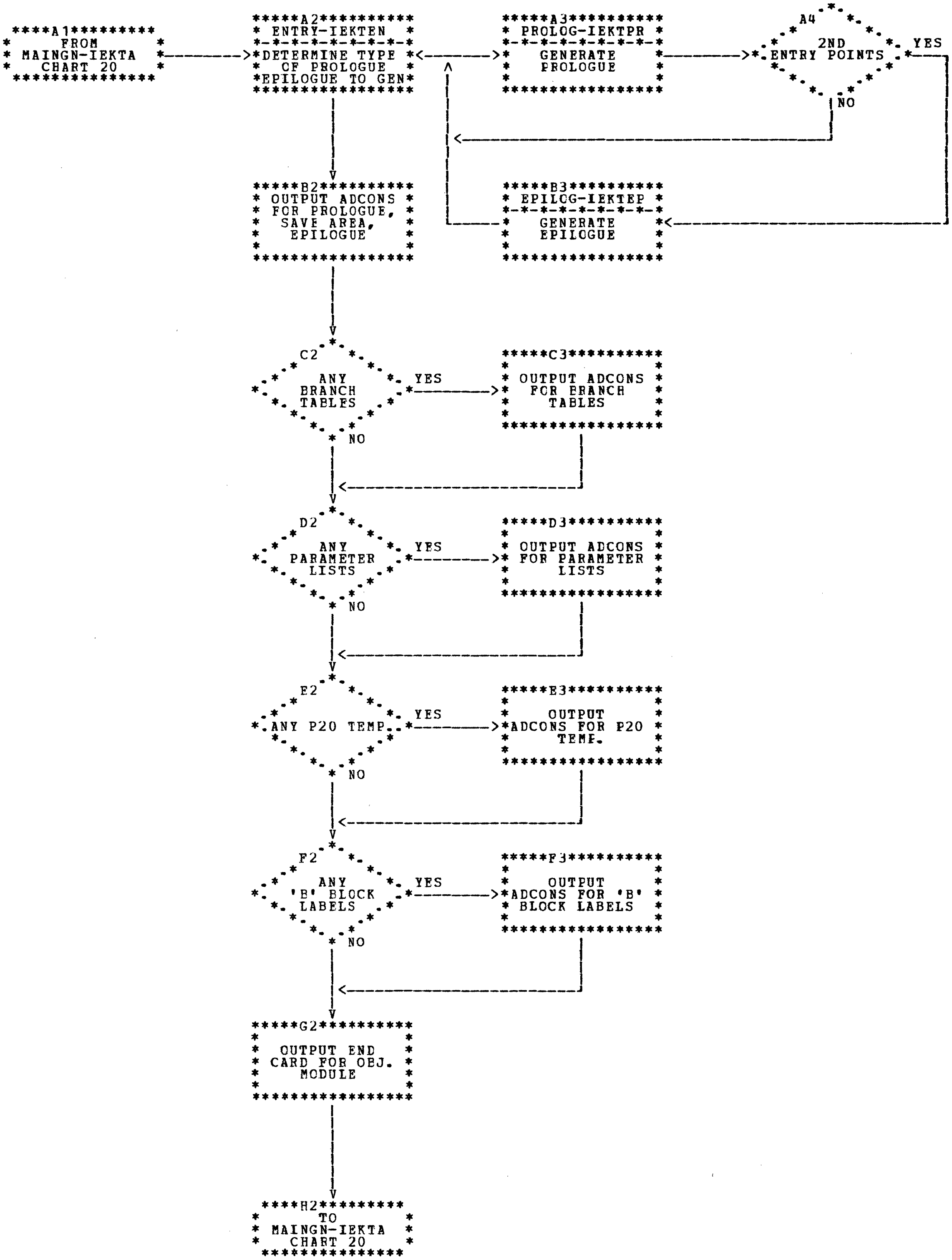
•Table 13. Phase 20 Utility Subroutines

Subroutine	Function
CIRCLE-IEKQCL (FOLLOW-IEKQF) *	Examines composit 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, adjusts text entry for possible mode change.
GETDIK-IEKPGK (FILTEX-IEKPFT) * (GETDIC-IEKPGC) * (INVERT-IEKPIV) *	Fills text space according to the arguments, gets space for temporaries, gets space for constants, obtains previous text entry.
KORAN-IEKQKO (LORAN-IEKQLO) *	Performs bit manipulation for text optimization, updates composit LMVS and LMVF matrixies.
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	Structured source program listing routine.
SUBSUM-IEKQSM	Replaces operands with equivalent values and, if possible, operand values with equivalent values.
WRITEX-IEKQWT	Diagnostic trace printing routine for text optimization.
XSCAN-IEKQXS (YSCAN-IEKQYS) * (ZSCAN-IEKQZS) *	Performs local block scan for backward movement, for common expression elimination, and for forward movement.
*Secondary entry point	

• Chart 20. Phase 25 Processing



• Chart 21. Subroutine END-IEKUEN



• Table 14. Phase 25 Subroutine Directory

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: BITCN, 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 GC TC operators, computed GO TO operators, unconditional branching, branch true and branch false operations, and ASSIGN statement.
CGNDTA-IEKWCN	Initializes the arrays used during code generation.
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 CALLs (other than those to IHCFCOMH) and function references. Generates the instructions to store the result returned by a function subprogram.
GOTOKK-IEKWKK	Used by MAINGN-IEKTA to branch to the code generation subroutines.
IOSUB-IEKTIS/ IOSUB2-IEKTIO	Generate 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 labels and branch table entries. 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, subtraction operations, half- and full-word integer multiplication, and half- and full-word integer division.
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[1]	Generates character strings in calls to IHCFCOMH for STOP and PAUSE statements.
SUBGEN-IEKVSU[1]	Generates instructions for the following text entries: subscript operations, right and left shift operations, store operations, and list item operations.

(Continued)

• Table 14. Phase 25 Subroutine Directory (Continued)

Subroutine	Function
TENTXT-IEKVTN	Controls the processing of END, RETURN, and I/O 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.

• Chart 22. Phase 30 (IEKP30) Overall Logic

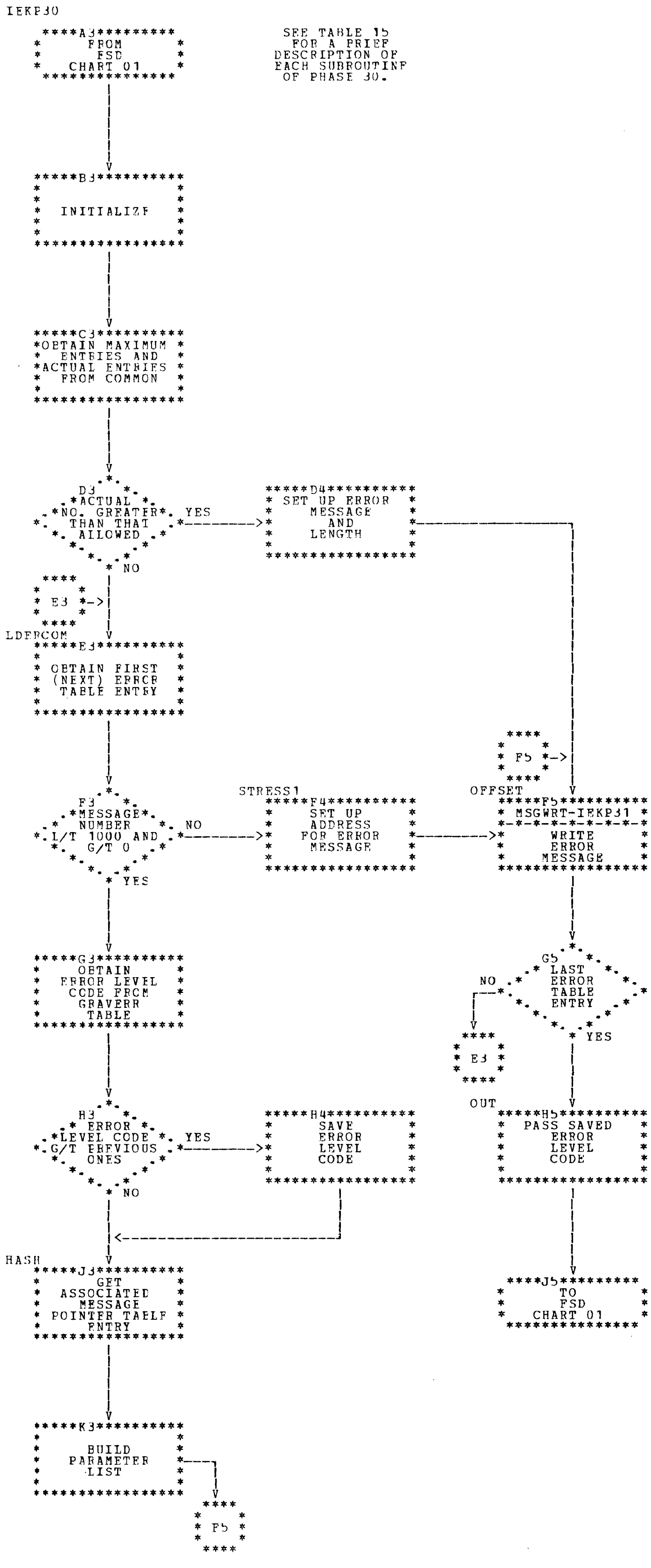


Table 15. Phase 30 Subroutine Directory

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

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 table and the keyword table. They are illustrated in Tables 17 and 18, respectively.

If the source statement has not been signaled as arithmetic during source statement packing (see note), 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(2,35))

1	Pointer to temporary for FLOAT/FIX	Pointer to 1-character symbol chain
2	Previous classification code (phase 10); Reg used on last ARITH (phase 20,OPT=0)	Pointer to 2-character symbol chain
3	Options: SOURCE, MAP, ID, EDIT, LOAD, DECK, LIST, BCD, XREF	Pointer to 3-character symbol chain
4	Pointer to most recently generated EQUIVALENCE group entry (phase 10); Relative location of first temporary (phase 20)	Pointer to 4-character symbol chain
5	NADCON index for first temporary (phase 20)	Pointer to 5-character symbol chain
6	Maximum line count	Pointer to 6-character symbol chain

(Continued)

• Table 16. Communication Table (NPTR(2,35)) (Continued)

7	NADCON index for last statement number	Pointer to last dictionary entry in stmt number chain (XREF-phase 10); Number of registers reserved for RX branches (phases 20 and 25)
8	Type of text (phase 10); Pointer to next phase 10 text item (phase 15); Pointer to .QXX temporary chain (phase 20)	
9	Pointer to next available phase 10 text entry	Pointer to last available phase 10 text entry
10	Name of routine (subprogram/main program)	
11	Phase in control indicator	Trace switch; optimization downgrade switch
12	Last error table entry	
13	END card indicator (phase 10)	Pointer to first card of source pgm
14	Pointer to parameters	Pointer to 4-byte constant chain
15	NADCON index for first parameter list	Pointer to 8-byte constant chain
16	Page count	Pointer to 16-byte constant chain
17	Current line count	Pointer to statement number chain
18	Relative location for register 13	Number of branch table entries; relative location of register 12
19	Active register: zero for reg 13, nonzero for Reg 12	NADCON index for statement number adcons
20	Secondary entry points if nonzero	Number of times XREF buffer has been written out (phase 10)
21	Location counter	NADCON index for first COMMON area
22	Pointer to dictionary entry for IBCOM	Next available error table entry
23	External function and/or CALL indicator	Pointer to end of statement number chain
24	Program uses FLOAT/FIX or MOD function if nonzero; arithmetic interrupt indicator	Optimization level
25	Pointer to first dictionary entry	Pointer to common chain
26	Pointer to DEFINE FILE text	Pointer to equivalence chain
27	Pointer to literal constant chain	Pointer to data text chain
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	Pointer to format text chain
33	Size of constants	Size of variables
34	Current displacement from active register (phase 20)	Adcon entry number
35	Relative location for first statement number	Delete/error switch

(Continued)

• Table 17. Keyword Pointer Table

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 key words beginning with the associated character.
[2]This field contains the displacement from the beginning of the key word table for the group of key words associated with character.

• Table 18. Keyword Table

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 (group mark) *	23
4	ENTRY	22
7	EXTERNAL	20
5	FORMAT	25
7	FUNCTION	24
3	FIND	12
3	GOTO	27
7	IMPLICIT	29
14	INTEGERFUNCTION	28
6	INTEGER	30
14	LOGICALFUNCTION	33

*Represented in hex as 'C5D5C44F'

(Continued)

•Table 18. Keyword Table (Continued)

Length-1[1]	Key Word[2]	Code[3]
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.

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 19.

•Table 19. 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 as shown in 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 order 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 (refer to the section, "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.

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 reentered; if 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 reentered: if 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

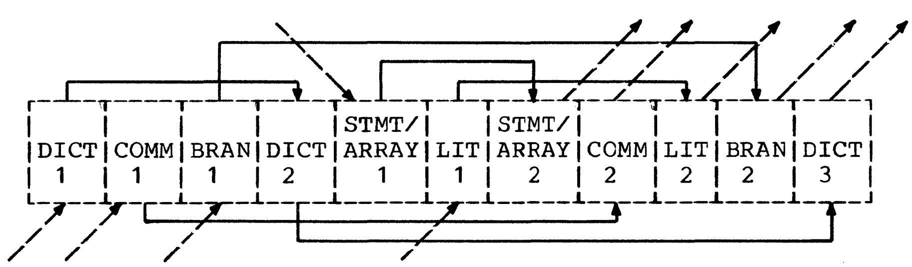


Figure 10. Information Table Chains

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 mid-point), for the entries describing variables of the same length or constants of the same size. The initial mid-point 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 mid-point, the first entry on the current mid-point's high chain becomes the new mid-point. Similarly, when two entries have been made on the low side of the mid-point, the first entry on the current mid-point's low chain becomes the new mid-point.

A change of mid-point for a variable of a particular length or a constant of a particular size causes a pointer to the new mid-point 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 (refer to "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 (refer to "Communication Table"). The entry for D is the initial mid-point 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 mid-point and compares C to the variable in that entry. If the two are unequal, phase 10 determines if 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.

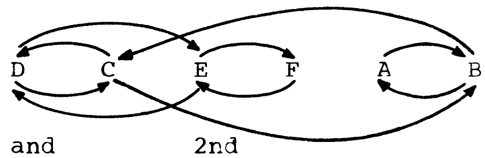
When the variable E is encountered, phase 10 carries out essentially the same procedure; however, because E is greater than D, phase 10 examines the high chain field of the entry for D. It is zero, which denotes the end of the chain. Phase 10 therefore places the dictionary entry for E into the next 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 initial mid-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 mid-point, the first variable on D's high chain becomes the new mid-point.

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.



1st and 3rd mid-points

2nd mid-point

Note: High and low chains are maintained for all entries. When the entry for F is made, the mid-point shifts from D to E. When the entry for A is made, the mid-point shifts from E to D.

Figure 11. Dictionary Chain

Statement Number Chain Operation

The statement number chain constructed by phase 10 is linear; that is, each statement number entry (refer to "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 order 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 only entered once.)

Common Chain Operation

The chain constructed by phase 10 for the common information appearing in the source module is bi-linear; that is, phase 10 links together:

1. The individual common block name entries (refer to "Common Table") that it develops for the common block names appearing in the module.
2. The dictionary entries (refer to "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 for the equivalence information appearing in the source module is also bi-linear. Phase 10 links together:

1. The individual equivalence group entries (refer to "Common Table") that it constructs for the equivalence groups appearing in the module.
2. The equivalence variable entries (refer to "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 which 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 (refer to "Branch Table") it places into the information table in the appropriate field of the communication table. For each other branch table entry, phase 10 records a pointer to its location in the information table in the chain field of the previously developed branch table entry. Unlike statement number entry processing, no label comparison is necessary. Scanning the chain is therefore 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 figures illustrating the phase 10 formats of the entries of each components. Modifications made to these entries by subsequent phases of the compiler are also illustrated in figure form.

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.

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.

4 bytes

	High chain field	
Byte A usage field	Byte B usage field	DIS field
	Low chain field	
Mode field		Type field
Used by STALL-IEKGST	P1 field	
Not used		
SF field	Common chain field	
	Name field	
	Name field	

Figure 12. Format of Dictionary Entry for Variable

Subfield	Function
Bit 0 'on'	variable is structured
Bit 1 'on'	symbol referred to
Bit 2 'on'	variable is in common
Bit 3 'on'	not used
Bit 4 'on'	variable is equated
Bit 5 'on'	variable has appeared in an equivalence group that has been processed by STALL-IEKGST (used by phase 15)
Bit 6 'on'	variable is an external function name
Bit 7 'on'	not used

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

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'	not used
Bit 4 'on'	variable has appeared in a previous DATA statement (phase 15)
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 20 and 21).

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.

• Table 20. Operand Modes

Mode of Operand	Internal Representation (in hexadecimal)
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 21. Operand Types

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

SF Field: The SF field contains STORE-FETCH information for the variable. If the variable is stored into, bit 0=1; if the variable if fetched, bit 1=1.

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 CSORN-IEKCCR processing is illustrated Figure 15.

← 4 bytes →

	*	
*	*	*
	*	
*		*
*	*	
*		
*	*	
XREF buffer pointer-last entry		*
*		
XREF buffer count		XREF buffer pointer-first entry

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

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 the time this dictionary entry is modified by CSCRN-IEKCCR.

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

Dictionary Entry for Variable After Dictionary Rechaining: The format of a dictionary entry for a variable after the dictionary has been rechained during STALL-IEKGST is illustrated in Figure 16.

```
<--------------- 4 bytes --------------->
+----------+------------------------------+
|          |       New chain field        |
+----------+---------+--------------------+
|*         |*        |*                   |
+----------+---------+--------------------+
|          |*                             |
+----------+---------+--------------------+
|*                   |*                   |
+----------+---------+--------------------+
|*         |*                             |
+----------+------------------------------+
|*                                        |
+-----------------------------------------+
|*                                        |
+-----------------------------------------+
|*                                        |
+-----------------------------------------+
|*                                        |
+-----------------------------------------+
```

• Figure 16. Format of Dictionary Entry for Variable After Rechaining

Dictionary Entry for Variable After Coordinate Assignment: The format of a dictionary entry for a variable after coordinate assignment by STALL-IEKGST is illustrated in Figure 17.

```
<--------------- 4 bytes --------------->
+----------+------------------------------+
|          |*                             |
+----------+---------+--------------------+
|*         |*        |*                   |
+----------+---------+--------------------+
|          |*                             |
+----------+---------+--------------------+
|*                   |*                   |
+----------+---------+--------------------+
|Coordinate|*                             |
|number for|                              |
|variable  |                              |
+----------+------------------------------+
|*                                        |
+-----------------------------------------+
|*                                        |
+-----------------------------------------+
|*                                        |
+-----------------------------------------+
|*                                        |
+-----------------------------------------+
```

• Figure 17. 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 18.

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 19.

```
<--------------- 4 bytes --------------->
+----------+------------------------------+
|          |       New chain field        |
+----------+---------+--------------------+
|*         |*        |Displacement from   |
|          |         |start of common     |
|          |         |block               |
+----------+---------+--------------------+
|          |*                             |
+----------+---------+--------------------+
|*                   |*                   |
+----------+---------+--------------------+
|*         |*                             |
+----------+------------------------------+
|*                                        |
+-----------------------------------------+
|*                                        |
+-----------------------------------------+
|*                                        |
+-----------------------------------------+
|*                                        |
+-----------------------------------------+
```

• Figure 18. Format of Dictionary Entry for Variable After Common Block Processing

```
<--------------- 4 bytes --------------->
+----------+------------------------------+
|          |       New chain field        |
+----------+---------+--------------------+
|*         |*        |Displacement from   |
|          |         |start of common     |
|          |         |block               |
+----------+---------+--------------------+
|          |Pointer to entry containing   |
|          |pointer to address constant   |
|          |for variable                  |
+----------+---------+--------------------+
|*                   |*                   |
+----------+---------+--------------------+
|*         |*                             |
+----------+------------------------------+
|*                                        |
+-----------------------------------------+
|*                                        |
+-----------------------------------------+
|*                                        |
+-----------------------------------------+
|*                                        |
+-----------------------------------------+
```

• Figure 19. Format of Dictionary Entry for a Variable After Relative Address Assignment

CONSTANT ENTRY FORMAT: The format of the dictionary entries constructed by phase 10 for the constants of the source module is illustrated in Figure 20.

The format of a dictionary entry for a constant is the same as for a variable. The changes the entry undergoes during processing are the same except that bytes 3 and 4 of word two contain a displacement from an associated address constant and a constant does not undergo XREF processing. Also, for constants referred to implicitly, PHAZ15 sets a referenced bit on. (bit 1 in byte A usage field), see Figure 13.

← 4 bytes →

	Chain field	
Byte A Usage field	Byte B Usage field	Used by phase 15
Mode field		Type field
Used by STALL-IEKGST	zero	
Constant field		
Constant field		
Constant field		
Constant field		

Figure 20. Format of Dictionary Entry for Constant

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 21.

← 4 bytes →

	Chain Field		
Byte A Usage	Byte B Usage	Used by phase 20	Used by phase 20
	Pointer field		
Image field			
Used by phase 15			
Used by phase 15			

Figure 21. Format of a Statement Number Entry

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.

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 22 indicates the function of each subfield of this field.

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

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 23 indicates the function of each subfield in the byte B usage field.

Pointer Field: This field contains a pointer to the text entry constructed by phase 10 for the associated statement number.

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

MODIFICATIONS TO STATEMENT NUMBER ENTRIES: During the processing of subroutine 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, LABTLU-IEKCLT makes changes in statement number dictionary entries for later use by XREF-IEKXRF. (See Figure 24.)

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
Bit 7 'on'	statement number is used in a computed GO TO statement

Figure 23. Function of Each Subfield in the Byte B Usage Field of a Statement Number Entry

```
<--------------- 4 bytes --------------->
+--------+-------------------------------+
|        | *                             |
+--------+---------+---------+-----------+
|*       | *       |*        |*          |
+--------+---------+---------+-----------+
|        | *                             |
+--------+-------------------------------+
|*                                       |
+----------------------------------------+
|XREF buffer pointer - last entry        |
+------------------+---------------------+
|XREF buffer count |XREF buffer pointer- |
|                  |first entry          |
+------------------+---------------------+
|Definition field                        |
+----------------------------------------+
|*                                       |
+----------------------------------------+
|Sequence chain field                    |
+----------------------------------------+
```

• Figure 24. Format of a Dictionary Entry for Statement Number After 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 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 25 illustrates the format of a statement number entry after the processing of STALL-IEKGST and phases 15, 20, and 25. Only changes are indicated; * stands for unchanged.

```
<------------------- 4 bytes ------------------->
+----------+-------------------------------------+
|          |New Chain field                      |
+----------+---------------+---------------------+
|*         |*       |Block        |Loop          |
|          |        |Status       |number        |
|          |        |Field        |              |
+----------+--------+-------------+--------------+
|          |Address constant pointer field       |
+----------+-------------------------------------+
|          Image field                           |
+----------+-------------------------------------+
|Loop      |Text pointer field                   |
|number    |                                     |
|save area |                                     |
+----------+-------------------------------------+
|Forward connection field (ILEAD)                |
+------------------------------------------------+
|Backward connection field (JLEAD)               |
+------------------------------------------------+
|Block size field (BSZ)                          |
+------------------------------------------------+
```

• Figure 25. 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 the statement number that is defined in the source module immediately after the statement number for which the statement number entry under consideration was constructed. (STALL-IEKGST 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 26 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 26. 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:

- An indication of a reserved register and a displacement, if branching optimization is being implemented and if the text block (associated with the statement number entry under consideration) can be branched to via an RX-format branch instruction (refer to the phase 20, "Branching Optimization").

- A pointer to the address constant reserved for the statement number (refer to phase 25, "ADCON Table Entry Reservation").

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. A relative address of the block is stored in this field by phase 20.

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 a relative location is stored in the field.

DIMENSION ENTRY FORMAT: The format of the dimension entries constructed by phase 10 is illustrated in Figure 27.

◄──── 4 bytes ────►

Array size field		
Dimension number field		Element length field
	First subscript pointer field	
	Second subscript pointer field	
	Third subscript pointer field	
	Fourth subscript pointer field	
	Fifth subscript pointer field	
	Sixth subscript pointer field	
	Used only for variable dimensions	

• Figure 27. Format of Dimension Entry

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.

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, (Refer to "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.

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.

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.

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.

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.

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.

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.

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 28.

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.

<table>
<tr><td colspan="2" align="center">← 4 bytes →</td></tr>
<tr><td></td><td>Chain field</td></tr>
<tr><td></td><td>P1 field</td></tr>
<tr><td></td><td>P2 field</td></tr>
<tr><td colspan="2">Name field</td></tr>
<tr><td colspan="2">Name field</td></tr>
<tr><td>Character Number field</td><td>ISN field</td></tr>
</table>

Figure 28. 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 29 illustrates the format of a common block name entry after common block processing by STALL-IEKGST. Only changes are indicated; * stands for unchanged.

← 4 bytes →

<table>
<tr><td></td><td colspan="2">*</td></tr>
<tr><td></td><td colspan="2">*</td></tr>
<tr><td></td><td colspan="2">Total size of common block</td></tr>
<tr><td colspan="3">*</td></tr>
<tr><td colspan="3">*</td></tr>
<tr><td colspan="2">*</td><td>*</td></tr>
</table>

Figure 29. 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 30.

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.

← 4 bytes →

<table>
<tr><td>Indicator field</td><td>Chain field</td></tr>
<tr><td colspan="2">P1 field</td></tr>
<tr><td colspan="2">Used by phase 15</td></tr>
<tr><td colspan="2">ISN field</td></tr>
</table>

Figure 30. 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 31 illustrates the format of an equivalence group entry after equivalence processing by STALL-IEKGST. Only changes are indicated; * stands for unchanged.

← 4 bytes →

<table>
<tr><td>*</td><td></td></tr>
<tr><td colspan="2">*</td></tr>
<tr><td colspan="2">Pointer to the "head" of the equivalence group</td></tr>
<tr><td colspan="2">*</td></tr>
</table>

Figure 31. 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 32.

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.

← 4 bytes →

<table>
<tr><td>Indicator field</td><td>P1 field</td></tr>
<tr><td>Number of subscripts</td><td>Chain field</td></tr>
<tr><td colspan="2">Offset field</td></tr>
<tr><td colspan="2">Subscript field</td></tr>
<tr><td colspan="2">Subscript field</td></tr>
</table>

Figure 32. Format of Equivalence Variable Entry

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 33 illustrates the format of an equivalence variable entry after equivalence processing by STALL-IEKGST. Only changes are indicated; * stands for unchanged.

◄—— 4 bytes ——►

*	*
	*
Displacement of variable from group head	
*	
*	

• Figure 33. Format of Equivalence Variable Entry After Equivalence Processing

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 34.

◄—— 4 bytes ——►

Chain field		
Length field	255	Used by phase 15
Literal constant field		

• Figure 34. 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 35 illustrates the format of a literal constant entry after relative address assignment by CORAL, the second segment of phase 15. Only changes are indicated; * stands for unchanged.

◄—— 4 bytes ——►

*		
*		Displacement from associated address constant
*		

• Figure 35. Format of Literal Constant Entry After Relative Address Assignment

LITERAL DATA ENTRY FORMAT: The format of the literal data entries constructed by phase 10 is illustrated in Figure 36.

Length field	(1 byte)
Literal data field	(1-255 bytes)

• Figure 36. 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 Table

The branch table contains 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 37.

4 bytes

Indicator field	Chain field
P1 field	
Used by phase 25	
Used by phase 25	

Figure 37. 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 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 38 illustrates the format of an initial branch table entry after the processing of phase 25 is complete. Only changes are indicated; * stands for unchanged.

4 bytes

*	*
*	
Relative address of statement associated with fall-through statement number	
Pointer to address constant reserved for fall-through statement number	

Figure 38. 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 39 illustrates the format of a standard branch table entry after the processing of phase 25 is complete. Only changes are indicated; * stands for unchanged.

4 bytes

*	*
*	
Relative address of statement associated with this statement number	

Figure 39. Format of Standard Branch Table Entry After Phase 25 Processing

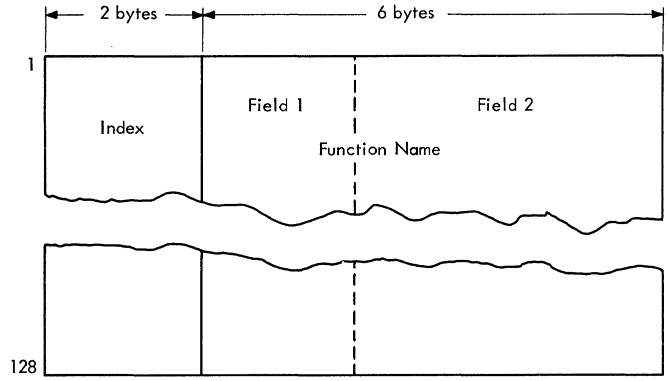
SUBPROGRAM TABLE

The subprogram table (IEKLFT) contains entries for the IBM supplied 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 subprogram table is used by phase 15 to determine the validity of the arguments to the subprogram.

Each entry in the subprogram table (see Table 22) contains two fields: index field (2 bytes) and 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 22. Subprogram Table - IEKLFT (2,128)



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 subprogram 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 on, then the operand 1 position of a text entry having that operator contains a valid or actual operand. If the bit is 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 23. 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 23. 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										
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 (refer to Table 24), global assignment tables (refer to Table 26), 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 sixteen words long. Words 1 through 11 represent general registers 1 through 11, words 12, 14, and 16 represent floating point registers 2, 4 and 6, and words 13 and 15 are unused.

• Table 24. 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.	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 25. BVA Table

Bit	Meaning
0	Not used.
1	Text item is candidate for forward movement.
2	Not used.
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 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	P1 is the result of an integer multiply or divide.
13	Zero length temporary indicator.
14	Case II subscript indicator is changed to a Case II.
15 . . 31	BVA - Local Activity.

The BVA table consists of a fullword for each text in the block.

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).

Table 26. 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 MCCORD 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 MCCORD 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 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.

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 (refer to Fortran System Director, "Generation of Initialization Instructions"). Register 13 is also used to refer to:

- Branch tables for computed GO TOs.
- 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. It is set aside to contain the starting address of the "Constants" portion of text information.

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. (Refer to 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 NAMELISTs. 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 40.

Name field (2 words)

Figure 40. 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 41.

<table>
<tr><td colspan="3">Name field (2 words)</td></tr>
<tr><td colspan="3">Address field (1 word)</td></tr>
<tr><td>Item Type field (1 byte)</td><td>Mode field (1 byte)</td><td>Not used (2 bytes)</td></tr>
</table>

Figure 41. 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 42.

Name field (2 words)			
Address field (1 word)			
Item Type field (1 byte)	Mode field (1 byte)	Number of dimensions field (1 byte)	Element length field (1 byte)
Indicator field (1 byte)	First dimension factor field (3 bytes)		
Not used (1 byte)	Second dimension factor field (3 bytes)		
Not used (1 byte)	Third dimension factor field (3 bytes)		
Etc. (refer to "Dimension Entry Format")			

Figure 42. 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 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; the second 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 (H) Programmer's Guide).

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.
- Format text - the internal representations of FORMAT 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.

Note: Intermediate text representations are, for sub-block 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 sub-blocks of main storage. The sub-blocks that constitute an intermediate text area are obtained by phase 10, as needed, via requests to the FSD (see FORTRAN System Director, "Storage Distribution").

Intermediate Text Chains

Each intermediate text area (i.e., the sub-blocks allocated to a particular type of text) is arranged as a chain, which 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 in 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 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 comprising the intermediate text representation of a FORMAT statement are not chained.)

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.

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 43 illustrates the format of an intermediate text entry constructed by phase 10.

◄—— 4 bytes ——►

Adjective code field (operator)	Chain field	
Mode field		Type field
0	Pointer field (operand)	

•Figure 43. 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 27). 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 20 and 21).

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 of the above form. FORMAT text entries consist of the characters of the FORMAT statement in source form packed into successive text entries.

•Table 27. Adjective Codes

Code (in decimal)	Mnemonic (where applicable)	Meaning
1	.NOT.	NOT
4	.AND.	AND
5	)	Right arithmetic parenthesis
6	.OR.	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
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

(Continued)

• Table 27. Adjective Codes (Continued)

Code (in decimal)	Mnemonic (where applicable)	Meaning
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
223	GLDF	Generated statement number definition
225		WRITE using NAMELIST

(Continued)

•Table 27. Adjective Codes (Continued)

Code (in decimal)	Mnemonic (where applicable)	Meaning
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 44.

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 44. 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 45.

Adjective Code	Chain	Mode	Type	0	Pointer
DATA		0	0		→ 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 45. 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 46.

Adjective Code	Chain	Mode	Type	0	Pointer
NAMELIST		NAMELIST	0		→ NAME1
/		0	0		→ To text for next NAMELIST block
LIST		Real	Array		→ A
LIST		Real	Scalar		→ B
LIST	0	Real	Scalar		→ C
NAMELIST		NAMELIST	0		→ NAME2
/		0	0		→ To text for next NAMELIST block
LIST		Real	Scalar		→ D
LIST		Real	Scalar		→ E
LIST	0	Real	Array		→ F
NAMELIST		NAMELIST	0		→ NAME3
/		0	0		→ To text for next NAMELIST statement
LIST	0	Real	Scalar		→ G
1 byte	3 bytes	2 bytes	2 bytes	1 byte	3 bytes

• Figure 46. 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 I/O 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 47.

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 47. Phase 10 Define File 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		Statement number	0		5
FORMAT		0	0		To text for next FORMAT statement
1 byte	3 bytes	2 bytes	2 bytes	1 byte	3 bytes

(2H0 [2]	A,A6 [2]	//5X [2]	,3(I [2]		4,E1 [2]
2.5, [2]	3F12 [2]	.3,'[2]	ABC' [2]		))ǂ[1]

[1]Group mark.
[2]One character per byte.

• Figure 48. Phase 10 Format 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 49.

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 49. Phase 10 SF Skeleton Text

PHASE 15/PHASE 20 INTERMEDIATE TEXT MODIFICATIONS

During phase 15 and phase 20 text processing, the intermediate text entries are modified to a form 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 processed by phase 15. Unchanged text is passed on to subsequent phases in phase 10 format with but one modification: the contents of the operator and chain fields are switched.

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 50.

←4 bytes→

Indicator field	Chain field
P1 field	
P2 field	
Offset field	
Number field	

Figure 50. Format 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 51 indicates the function of each subfield in the indicator field.

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 Hollerith 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 51. Function of Each Subfield in Indicator Field of Phase 15 Data Text Entry

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 52.

← 4 bytes →

Chain field		
	Operator field	Indicator field
P1 field		
BLKEND field		
Use vector field (MVF) (4 words)		
Definition vector field (MVS) (4 words)		
Busy-on-exit vector field (MVX) (4 words)		

•Figure 52. 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.

Operator Field: The operator field contains an internal operation code (numeric) for a statement number definition (see Table 28).

•Table 28. Phase 15/20 Operators

Code (in decimal)	Mnemonic (where applicable)	Meaning
1	.NOT.	NOT
2	U	Unary minus
4	.AND.	AND
5	)	Right parenthesis
6	.OR.	OR
7	.XOR.	XOR
8	ST	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

(Continued)

•Table 28. Phase 15/20 Operators (Cont.)

Code (in decimal)	Mnemonic (where applicable)	Meaning
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
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 zero
35	BGEZ	Branch greater than or equal zero
36	BLEZ	Branch less than or equal zero
37	BEZ	Branch equal to zero
39	NMLS	NAMELIST operands
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

(Continued)

•Table 28. Phase 15/20 Operators (Cont.)

Code (in decimal)	Mnemonic (where applicable)	Meaning
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
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

(Continued)

•Table 28. Phase 15/20 Operators (Cont.)

Code (in decimal)	Mnemonic (where applicable)	Meaning
78	BITOFF	BITOFF in-line routine
79	BITFLP	BITFLP in-line routine
80	ANDF	ANDF in-line routine
81	ORF	ORF in-line routine
82	COMPL	COMPL in-line routine
83	MOD24	MOD24 in-line routine
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
193		BLOCK DATA
200		COMMON
201		EQUIVALENCE
202		EXTERNAL
203		Register usage (phase 20 only)
205		DATA
208		FUNCTION
209		FORMAT
210		END I/O

(Continued)

•Table 28. Phase 15/20 Operator (Cont.)

Code (in decimal)	Mnemonic (where applicable)	Meaning
211		CONTINUE
212		Relative record number
213		Object time FORMAT
214		BACKSPACE
215		REWIND
216		END FILE
217		WRITE unformatted
218		READ unformatted
219		WRITE formatted
220		READ formatted
221		Begin I/O
222	LDF	Statement number definition
223	GLDF	Generated statement number definition
224		IMPLICIT
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
249	END	END
251		I/O unit number
252		FORMAT
253		NAMELIST

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 53 indicates the function of each subfield in the indicator field.

Subfield	Function
Bits 0-3	not used
Bit 4 'on'	associated block contains an I/O 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 53. 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 STALL-IEKGST are assigned a unique coordinate (1 bit) in this vector field. In general, if the ith bit is on (1), the variable or constant assigned to the ith coordinate is used in the associated block.

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 STALL-IEKGST are assigned a unique coordinate (1 bit) in this vector field. In general, if the ith bit is on (1), the variable assigned to the ith coordinate is defined in the associated block.

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 indicates which operands are busy-on-exit from the block. Variables and constants, as they are encountered by STALL-IEKGST are assigned a unique coordinate (1 bit) in this vector field. In general, during phase 15, if the ith bit is on (1), the variable assigned to the ith coordinate is not busy-on-entry to the block. During phase 20, if the ith bit is on, the variable or constant assigned to the ith coordinate is busy-on-exit from the block.

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 54.

<table>
<tr><td colspan="3">◄──── 4 bytes ────►</td></tr>
<tr><td colspan="3">Chain field</td></tr>
<tr><td>Set by phase 20
Used by phase 25</td><td>Operator field</td><td>Mode field</td></tr>
<tr><td>Set by phase 20
Used by phase 25</td><td colspan="2">P1 field</td></tr>
<tr><td>Set by phase 20
Used by phase 25</td><td colspan="2">P2 field</td></tr>
<tr><td>Set by phase 20
Used by phase 25</td><td colspan="2">P3 field</td></tr>
<tr><td colspan="3">Displacement field</td></tr>
</table>

•Figure 54. 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 28).

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 29.

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 form as the standard text output of phase 15. The format of the phase 20 output text is illustrated in Figure 55.

R1, R2, and R3 Fields: The R1, R2, and R3 fields (each is 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.

B1, B2, and B3 Fields: The B1, B2, and B3 fields (each is 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 30.

Table 29. Meanings of Bits in Mode Field of Standard Text Entry Status Mode Word

Mode	Bits	Meaning
general	27-28	00 - logical 01 - integer 10 - real
operand 1	29	0 - short mode (logical*1, integer*2, real*4) 1 - long mode (logical*4, integer, real*8)
operand 2	30	0 - short mode (logical*1, integer*2, real*4) 1 - long mode (logical*4, integer, real*8)
operand 3	31	0 - short mode (logical*1, integer*2, real*4) 1 - long mode (logical*4, integer, real*8)

◄——— 4 bytes ———►

<table>
<tr><td colspan="4">Chain field [1]</td></tr>
<tr><td colspan="2">Status field</td><td>Operator field [1]</td><td>Mode field [1]</td></tr>
<tr><td>R1</td><td>B1</td><td colspan="2">P1 field [1]</td></tr>
<tr><td>R2</td><td>B2</td><td colspan="2">P2 field [1]</td></tr>
<tr><td>R3</td><td>B3</td><td colspan="2">P3 field [1]</td></tr>
<tr><td colspan="4">Displacement field[1]</td></tr>
<tr><td colspan="4">[1]The chain field, mode field, operator field, P1 field, P2 field, P3 field, and displacement field are as defined in a phase 15 standard text entry. (Phase 20 does not alter these fields.)</td></tr>
</table>

•Figure 55. Format of Phase 20 Text Entry

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 28.

•Table 30. Status Field Bits and Their Meanings

Operand/ Base Address	Bit	Meaning
Operand 2 base address status	0-1	not used
	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	not used
	14	1 - divide item actually MOD function. If FC=44 or 15, load addresses precede.
	15	1 - .QXX temporary created for this item

Branch Operator (B)

←——— 4 bytes ———→

Chain				
Status			Branch operator	Mode
R1		P1		

P1: The P1 field contains a pointer to the statement number/array table entry for the statement number branched to.

Note: Phase 25 decides if an RR or an RX branch instruction should be generated.

Logical Branch Operators (BT, BF)

←——— 4 bytes ———→

Chain				
Status			Logical branch operator	Mode
R1		P1		
R2	B2	P2		

P1: The P1 field contains a pointer to the statement number/array table entry for the statement number being branched to.

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)

←——— 4 bytes ———→

Chain				
Status			Binary operator	Mode
R1	B1	P1		
R2	B2	P2		
R3	B3	P3		

Test and Set Operators (GT, LT, GE, LE, EQ, and NE)

```
◄──────────────────4 bytes──────────────────►
┌───────────────────────────────────────────┐
│ Chain                                     │
├─────────────────────────┬───────────┬─────┤
│ Status                  │Test and   │ Mode│
│                         │set        │     │
│                         │operator   │     │
├──────┬──────┬───────────┴───────────┴─────┤
│ R1   │B1    │P1                           │
├──────┼──────┼─────────────────────────────┤
│ R2   │B2    │P2                           │
├──────┼──────┼─────────────────────────────┤
│ R3   │B3    │P3                           │
└──────┴──────┴─────────────────────────────┘
```

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)

```
◄──────────────────4 bytes──────────────────►
┌───────────────────────────────────────────┐
│ Chain                                     │
├─────────────────────────┬───────────┬─────┤
│ Status                  │Function   │ Mode│
│                         │operator   │     │
├──────┬──────┬───────────┴───────────┴─────┤
│ R1   │B1    │P1                           │
├──────┼──────┼─────────────────────────────┤
│ R2   │B2    │P2                           │
├──────┼──────┼─────────────────────────────┤
│ R3   │B3    │P3                           │
├──────┼──────┼─────────────────────────────┤
│      │      │                             │
└──────┴──────┴─────────────────────────────┘
```

Testing a Byte Logical Variable (LDB)

```
◄──────────────────4 bytes──────────────────►
┌───────────────────────────────────────────┐
│ Chain                                     │
├─────────────────────────┬───────────┬─────┤
│ Status                  │LDB        │ Mode│
│                         │operator   │     │
├──────┬──────┬───────────┴───────────┴─────┤
│ R1   │B1    │                             │
├──────┼──────┼─────────────────────────────┤
│ R2   │B2    │                             │
├──────┼──────┼─────────────────────────────┤
│ R3   │B3    │                             │
└──────┴──────┴─────────────────────────────┘
```

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

4 bytes

Chain				
Status			Add operator	Mode
R1	B1	P1		
R2	B2	P2		
R3	B3	P3		

Text Entry 1

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.

4 bytes

Chain				
Status			Branch operator	Mode
R1		P1		
R2	B2	P2		
R3	B3	P3		

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 being branched to.

Computed GO TO Operator

4 bytes

Chain				
Status			Computed GO TO operator	Mode
R1		P1		
		P2		
R3		P3		

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)

← 4 bytes →

Chain				
Status			Branch	Mode
R1	B1	P1		
R2	B2	P2		
R3	B3	P3		

P1: The P1 field contains a pointer to the statement number/array table entry for the statement number being branched to.

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)

← 4 bytes →

Chain				
Status			Binary shift operator	Mode
R1	B1	P1		
R2	B2	P2		
		Shift quantity		

Load Address Operator (LA)

← 4 bytes →

Chain				
Status			Load address operator	Mode
R1	B1	P1		
R2	B2	P1		
R3	B3	P3		
Displacement				

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

←——— 4 bytes ———→

Chain			
Status		Subscript operator	Mode
R1	B1	P1	
R2	B2	P2	
R3	B3	P3	
Displacement			

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

←——— 4 bytes ———→

Chain			
Status		Subscript operator	Mode
		P1	
R2	B2	P2	
R3	B3	P3	
Displacement			

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)

←——— 4 bytes ———→

Chain			
Status		Operator	Mode
R1	B1	P1	
R2	B2	P2	
		Not used	

EXT and LIBF Operators

← 4 bytes →

Chain				
Status			Operator	Mode
R1	B1	P1		
R2	B2	P2		
R3	B3	P3		

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

← 4 bytes →

Chain				
Status			Argument operator	Mode
		P1		
		P2		
		P3 (for complex)		

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

← 4 bytes →

Chain				
Status			Argument operator	Mode
R1	B1			

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)

```
◄──────────────── 4 bytes ────────────────►
┌─────────────────────────────────────────┐
│ Chain                                   │
├────────────────────┬──────────┬─────────┤
│ Status             │Assigned  │Mode     │
│                    │GO TO     │         │
│                    │operator  │         │
├─────┬─────┬────────┴──────────┴─────────┤
│     │     │                             │
├─────┼─────┼─────────────────────────────┤
│ R2  │B2   │P2                           │
├─────┼─────┼─────────────────────────────┤
│     │     │                             │
└─────┴─────┴─────────────────────────────┘
```

P2: The P2 field contains a pointer to the variable being used in the assigned GO TC statement.

READ/WRITE Operators for I/O lists

READ

```
◄──────────────── 4 bytes ────────────────►
┌─────────────────────────────────────────┐
│ Chain                                   │
├────────────────────┬──────────┬─────────┤
│ Status             │READ      │Mode     │
│                    │operator  │         │
├─────┬─────┬────────┴──────────┴─────────┤
│ R1  │B1   │P1                           │
├─────┼─────┼─────────────────────────────┤
│     │     │                             │
├─────┼─────┼─────────────────────────────┤
│     │     │P3                           │
└─────┴─────┴─────────────────────────────┘
```

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 zero, an entire array is being read. This causes a different instruction sequence to be generated.

WRITE

```
◄──────────────── 4 bytes ────────────────►
┌─────────────────────────────────────────┐
│ Chain                                   │
├────────────────────┬──────────┬─────────┤
│ Status             │WRITE     │Mode     │
│                    │operator  │         │
├─────┬─────┬────────┴──────────┴─────────┤
│ R1  │B1   │                             │
├─────┼─────┼─────────────────────────────┤
│     │     │P2                           │
├─────┼─────┼─────────────────────────────┤
│     │     │P3                           │
└─────┴─────┴─────────────────────────────┘
```

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 zero, an entire array is being written. This causes a different instruction sequence to be generated.

Logical Branch Operators (BBT, BBF)

← 4 bytes →

<table>
<tr><td colspan="5">Chain</td></tr>
<tr><td colspan="3">Status</td><td>Logical Branch operator</td><td>Mode</td></tr>
<tr><td>R1</td><td></td><td colspan="3">P1</td></tr>
<tr><td></td><td>B2</td><td colspan="3">P2</td></tr>
<tr><td></td><td></td><td colspan="3">P3</td></tr>
</table>

P1: The P1 field contains a pointer to the statement number/array table entry for the statement number being branched to.

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

← 4 bytes →

<table>
<tr><td colspan="5">Chain</td></tr>
<tr><td colspan="3">Status</td><td>LBIT operator</td><td>Mode</td></tr>
<tr><td>R1</td><td>B1</td><td colspan="3">P1</td></tr>
<tr><td></td><td>B2</td><td colspan="3">P2</td></tr>
<tr><td></td><td></td><td colspan="3">P3</td></tr>
</table>

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 figures illustrate 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 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 below 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, zero may be changed to a one to perform the desired function. This typically 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.

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

IEKVTS: Used for the INT, IDINT, IFIX, and HFIX In-Line Routines

Index	Skeleton Instructions	INT, IFIX, HFIX Status	IDINT Status
		0011 0101	0011 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

---
[1]In some cases, operand 3 does not exist and only the status of operand 2 is indicated.

IEKVAD: Used for the ABS, IABS and DABS In-Line Routines

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

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

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	LNR 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

IEKVAD: Used for DMOD and AMOD In-Line Routines

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
	STD R1,Temp[1]	done by IEKVAD
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(0,12)	1111111111111111
10	MDR R1,R3	0111011101110111
11	MD R1,D(0,B3)	1000100010001000
12	LCDR R1,R1	1111111111111111
13	AD R1,D(0,B2)[1]	1111111100000000
14	ADR R1,R2	0000000011111111
15	L B1,D(0,BD)	XXXXXXXXXXXXXXXX
16	STD R1,D(0,B1)	XXXXXXXXXXXXXXXX

[1]When the statuses and base address statuses of operands 2 and 3 are zero, a store of operand 2 into a temporary will be done as indicated and the add will be from the temporary location.

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

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

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.

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

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(0,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(0,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 GC 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

| 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

| 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)	1111111111111111
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)	1111111111111111

| 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

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.

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

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

IEKVPL: Used for All Addition Operations and for Real Multiplication and Division 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)	1101000000000000
4	L B3,D(0,BD)	0000000000000000
5	LH R3,D(0,B3)	0100010001000100
6	LH R1,D(X,B3)	0000000000000000
7	LR R1,R2	0000110100001101
8	AR R1,R2	0000000000000000
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)	0000000000000000
13	STH R1,D(0,B1)	0000000000000000

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: 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

IEKVFP: Used for the LBIT, BBT, and BBF In-Line Routines

Index	Skeleton Instructions	BBT,BBF simple variable	BBT,BBF subscripted variable	LBIT simple variable	LBIT 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.

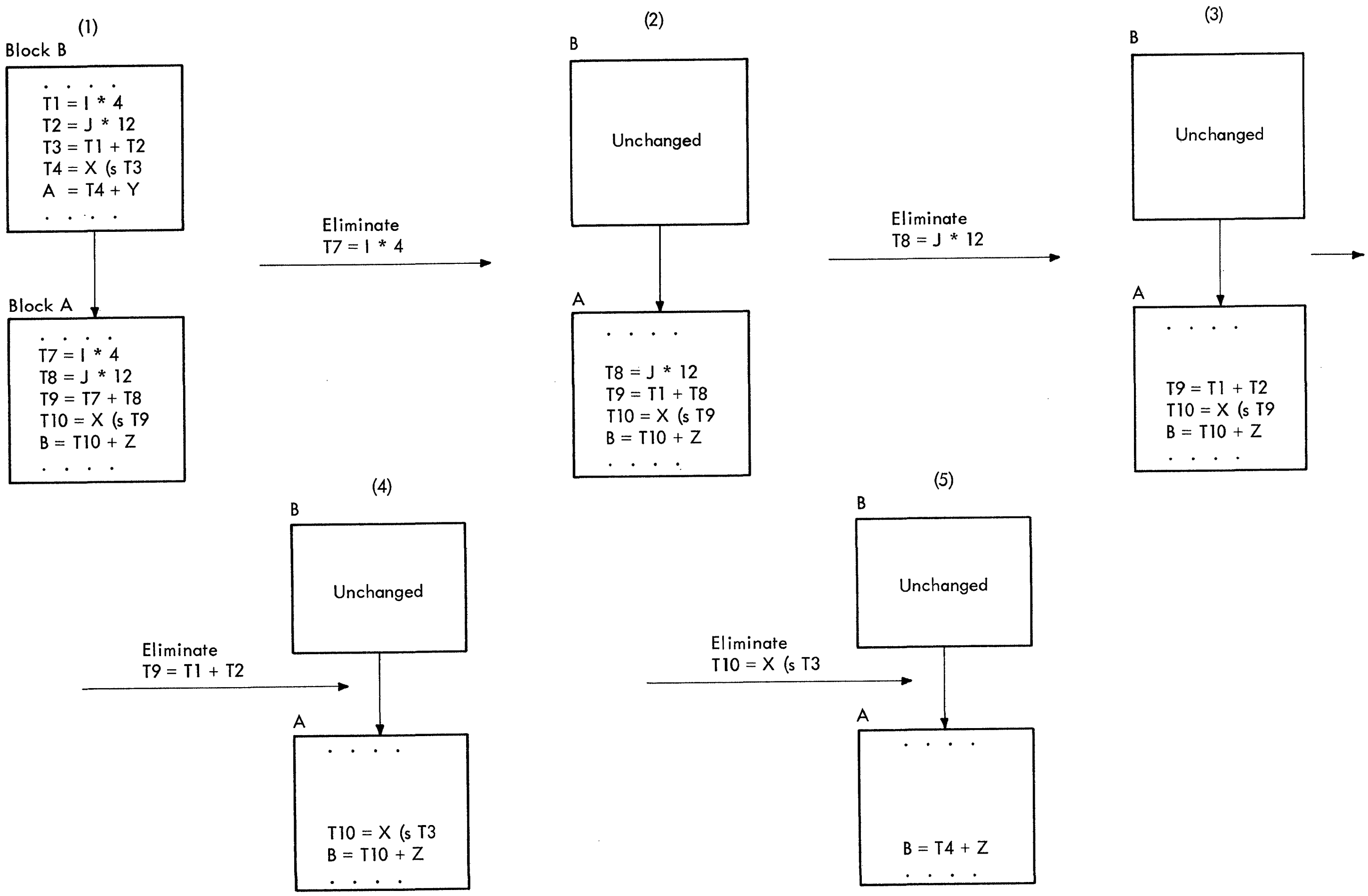
MM = 8 FOR BBF.

APPENDIX D: TEXT OPTIMIZATION EXAMPLES

This appendix contains examples that illustrate the effects of text optimization on sample text entry sequences. An example is presented for each of the five sections of text optimization.

Example 1: Common Expression Elimination

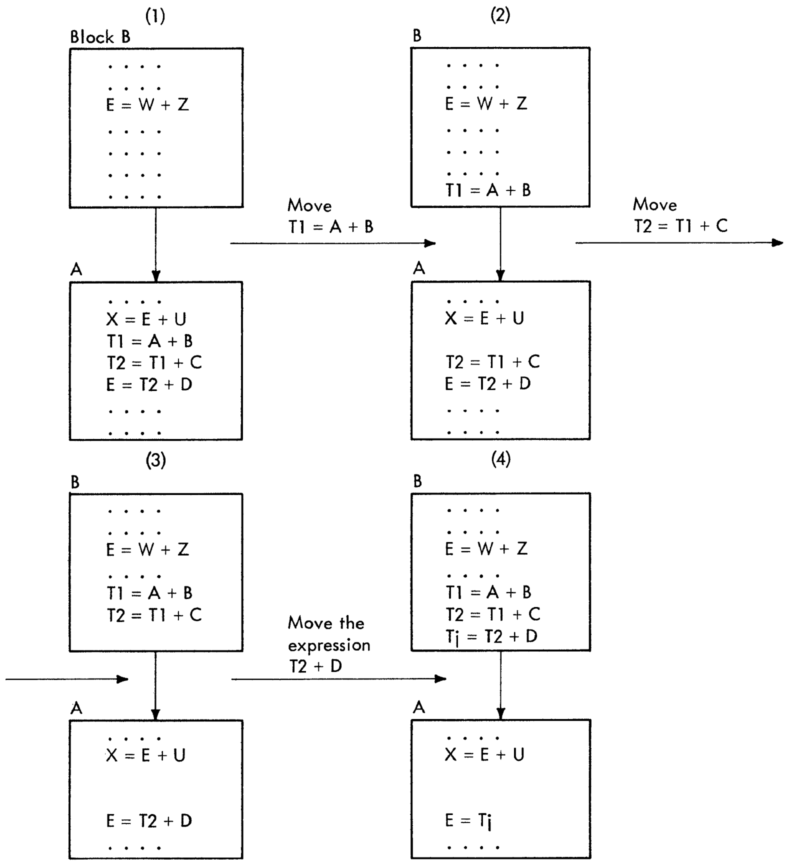
This example illustrates the concept of common expression elimination. The text entries in block A are to undergo common expression elimination. Block B is a back dominator of block A. Block B contains text entries that are common to those in block A.



NOTE: The items Ti are temporaries and (s represents a subscript operator

Example 2: Backward Movement

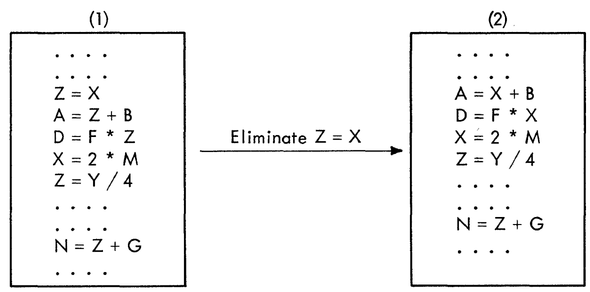
This example illustrates both methods of backward movement. The text entries in block A are to undergo backward movement. Block B is the back target of the loop containing block A.



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:

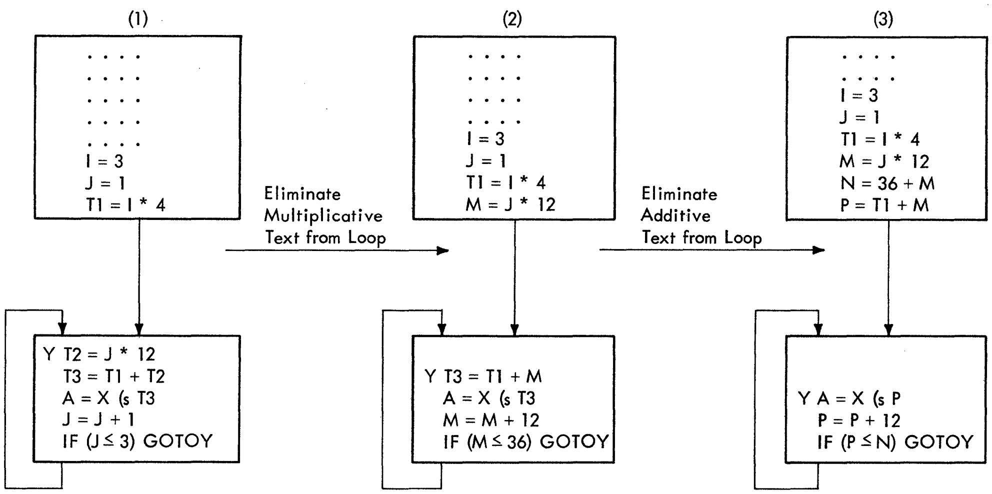
```
    I=3
    DO 10 J=1,3
    A=X(I,J)
10  CONTINUE
```

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 resided 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 resided 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 DC 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 form X(3,3) and that its elements are real (length 4).

The following figure illustrates the application of strength reduction to the loop.



This appendix describes the logic of some of the object-time library subprograms that may be referenced by the FORTRAN load module. Included at the end of this appendix are flowcharts that describe the logic of the subprograms.

Each object module, compiled from a FORTRAN source module, must be processed by the linkage editor prior to execution on the IBM System/360. The linkage editor must combine certain FORTRAN library subprograms with the object module to form an executable load module. The library subprograms exist as separate load modules on the FORTRAN system library (SYS1.FORTLIB). Each library subprogram that is externally referred to by the object module is included in the load module by the linkage editor. Among the library subprograms that may be so referred to are:

- IHCFCOMH (object-time I/O source statement processor) - entry name IBCOM#.
- IHCFIOSH (object-time sequential access I/O data management interface) - entry name FIOCS#.
- IHCNAMEL (object-time namelist routines) - entry names FRDNL# and FWRNL#
- IHCDIOSH (object-time direct access I/O data management interface) - entry name DIOCS#.
- IHCIBERH (object-time source statement error processor) - entry name IBERH#.
- IHCFCVTH (object-time conversion routine) - entry name ADCCN#.
- IHCDBUG[1] (object-time Debug Facility support routine) - entry name DEBUG#.
- IHCTRCH (object-time terminal error message and diagnostic traceback routine) - entry name IHCTRCH.
- IHCADST (object-time boundary adjustment routine) - entry name IHCADJST.

IHCFCOMH receives I/O requests from the FORTRAN load module via compiler-generated calling sequences. IHCFCOMH, in turn, submits these requests to the appropriate data management interface (IHCFIOSH or IHCDIOSH).

IHCFIOSH receives sequential access input/output requests from IHCFCOMH and, in turn, submits those requests to the appropriate BSAM (basic sequential access method) routines for execution.

IHCDIOSH receives direct access input/output requests from IHCFCOMH and, in turn, submits those requests to the appropriate BDAM (basic direct access method) routines for execution.

If source statement errors are detected during compilation, the compiler generates a calling sequence to the IHCIBERH subprogram. IHCIBERH processes object-time errors resulting from improperly coded source statements. IHCFCVTH contains the various object time conversion routines required by IHCFCOMH and IHCNAMEL. ICHTRCH processes terminal object-time error messages and produces a diagnostic traceback for IHCFCOMH. ICHADJST processes object time specification exceptions if the boundary alignment option is specified by the user during system generation.

IHCFCOMH

IHCFCOMH performs object-time implementation of the following FORTRAN source statements.

- READ and WRITE (for sequential I/O).
- READ, FIND, and WRITE (for direct access I/O).
- BACKSPACE, REWIND, and ENDFILE (sequential I/O device manipulation).
- STOP and PAUSE (write-to-operator).

In addition, IHCFCOMH: (1) processes object-time errors detected by various FORTRAN library subprograms, (2) processes arithmetic-type program interruptions, and (3) terminates load module execution.

All linkages from the load module to IHCFCOMH are compiler generated. Each time one of the above-mentioned source statements is encountered during compilation, the appropriate calling sequence to IHCFCOMH is generated and is included as part of the object module. At object-time,

----

[1]The FORTRAN IV (H) compiler does not have the code generation facilities for DEBUG statements. The discussion is included because the FORTRAN G compiler (which does include DEBUG) and the FORTRAN H compiler share a common library.

these calling sequences are executed, and control is passed to IHCFCOMH to perform the specified operation.

Note: IHCFCOMH itself does not perform the actual reading from or writing onto data sets. It submits requests for such operations to the appropriate I/O data management interface (IHCFIOSH or IHCDIOSH). The I/O interface, in turn, interprets and submits the requests to the appropriate access method (BSAM or BDAM) routines for execution. Figure 56 illustrates the relationship between IHCFCOMH and the I/O data management interfaces.

Charts 23, 24, and 25 illustrate the overall logic and the relationship among the routines of IHCFCOMH. Table 36, the IHCFCOMH routine directory, lists the routines used in IHCFCOMH and their functions.

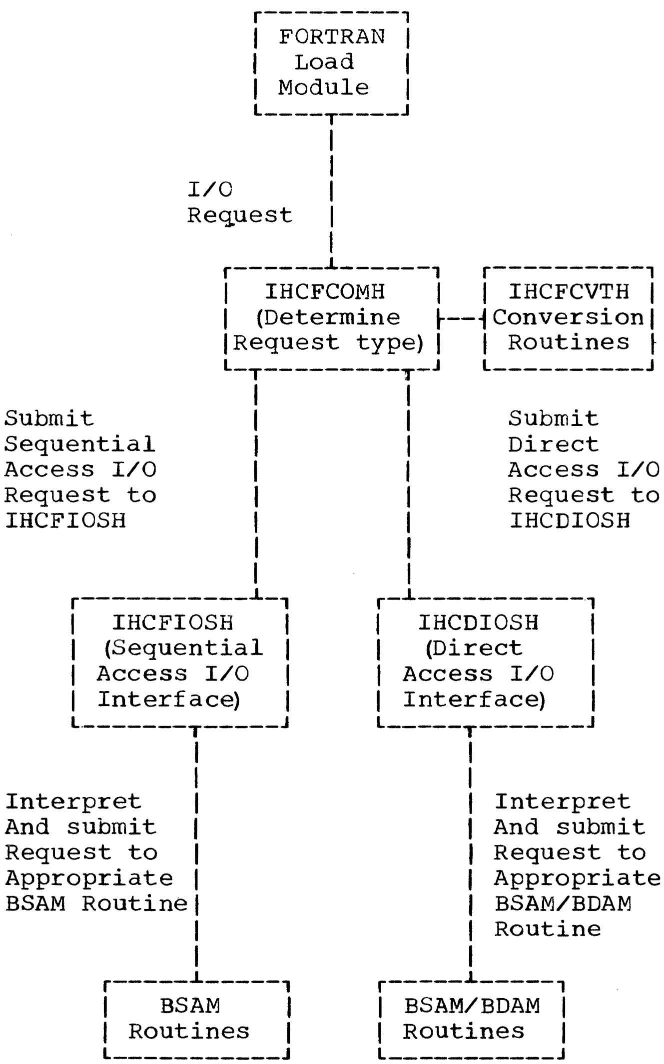


Figure 56. Relationship Between IHCFCOMH and I/O Data Management Interfaces

The routines of IHCFCOMH are divided into the following categories:

- Read/write routines.
- I/O device manipulation routines.
- Write-to-operator routines.
- Utility routines.

The read/write routines implement both the sequential I/O statements (READ and WRITE) and the direct access I/C statements (READ, FIND, and WRITE). (The direct access FIND statement is treated as a READ statement without format and list.)

The I/O device manipulation routines implement the BACKSPACE, REWIND, and END FILE source statements for sequential data sets. These statements are ignored for direct access data sets.

The write-to-operator routines implement the STOP and PAUSE source statements.

The utility routines: (1) process errors detected by FORTRAN library subprograms, (2) process arithmetic-type program interrupts, and (3) terminate load module execution.

READ/WRITE ROUTINES

The READ/WRITE routines of IHCFCOMH implement the various types of READ/WRITE statements of the FORTRAN IV language. For simplicity, the discussion of these routines is divided into two parts:

- READ/WRITE statements not using NAMELIST.

- READ/WRITE statements using NAMELIST.

READ/WRITE Statements Not Using NAMELIST

For the implementation of both sequential and direct access READ and WRITE statements, the read/write routines of IHCFCOMH consist of the following three sections:

- An opening section, which initializes data sets for reading and writing.
- An I/C list section, which transfers data from an input buffer to the I/O list items or from the I/O list items to an output buffer.
- A closing section, which terminates the I/O operation.

Within the discussion of each section, a read/write operation is treated in one of two ways:

- As a read/write requiring a format.
- As a read/write not requiring a format.

Note: In the following discussion, the term "read operation" implies both the sequential access READ statement and the direct access READ and FIND statements. The term "write operation" implies both the sequential access WRITE statement and the direct access WRITE statement.

OPENING SECTION: The compiler generates a calling sequence to one of four entry points in the opening section of IHCFCOMH each time it encounters a READ or WRITE statement in the FORTRAN source module. These entry points correspond to the operations of read or write, requiring or not requiring a format.

Read/Write Requiring a Format: If the operation is a read requiring a format, the opening section passes control to the appropriate I/O data management interface to initialize the unit number specified in the READ statement for reading. (The unit number is passed, as an argument, to the opening section via the calling sequence.) The I/O interface: (1) opens the data control block (via the OPEN macro instruction) for the specified data set if it was not previously opened, and (2) reads a record (via the READ macro instruction) containing data for the I/O list items into an I/O buffer that was obtained when the data control block was opened. The I/O interface then returns control to the opening section of IHCFCOMH. The address of the buffer and the length of the record read are passed to IHCFCOMH by the I/O interface. These values are saved for the I/O list section of IHCFCOMH. The opening section then passes control to a portion of IHCFCOMH that scans the FORMAT statement specified in the READ statement. (The address of the FORMAT statement is passed, as an argument, to the opening section via the calling sequence.) The first format code (either a control or conversion type) is then obtained.

For control type codes (e.g., an H format code or a group count), an I/O list item is not required. Control passes to the routine associated with the control code under consideration to perform the indicated operation. Control then returns to the scan portion, and the next format code is obtained. This process is repeated until either the end of the FORMAT statement or the first conversion code is encountered.

For conversion type codes (e.g., an I format code), an I/O list item is required. Upon the first encounter of a conversion code in the scan of the FORMAT statement, the opening section completes its processing of a read requiring a format and returns control to the next sequential instruction within the load module.

The action taken by IHCFCOMH when the various format codes are encountered is illustrated in Table 31.

If the operation is a write requiring a format, the opening section passes control to the I/O interface to initialize the unit number specified in the WRITE statement for writing. (The unit number is passed, as an argument, to the opening section via the calling sequence.) The I/O interface opens the data control block (via the OPEN macro instruction) for the specified data set if it was not previously opened. The I/O interface then returns control to the opening section of IHCFCOMH. The address of an I/O buffer that was obtained when the data control block was opened is saved for the I/O list section of IHCFCOMH. Subsequent opening section processing, starting with the scan of the FORMAT statement, is the same as that described for a read requiring a format.

Read/Write Not Requiring a Format: If the operation is a read or write not requiring a format, the opening section processing except for the scan of the FORMAT statement is the same as that described for a read or write requiring a format. (For a read or write not requiring a format, there is no FORMAT statement.)

I/O LIST SECTION: The compiler generates a calling sequence to one of four entry points in the I/O list section of IHCFCOMH each time it encounters an I/O list item associated with the READ or WRITE statement under consideration. These entry points correspond to a variable or an array list item for a read and write, requiring or not requiring a format. The I/O list section performs the actual transfer of data from: (1) an input buffer to the list items if a READ statement is being implemented, or (2) the list items to an output buffer if a WRITE statement is being implemented. In the case of a read or write requiring a format, the data must be converted before it is transferred.

Read/Write Requiring a Format: In processing a list item for a read requiring a format, the I/O list section passes control to the conversion routine associated with the conversion code for the list item. (The appropriate conversion routine is determined by the portion of IHCFCOMH that scans the FORMAT statement associated with the READ statement. The selection of the conversion routine depends on the conversion code of the list item being processed.)

Table 31. IHCFCOMH FORMAT Code Processing

FORMAT Code	Description	Type	Corresponding Action Upon Code by IHCFCOMH
	beginning of statement	control	Save location for possible repetition of the format codes; clear counters.
n(	group count	control	Save n and location of left parenthesis for possible repetition of the format codes in the group.
n	field count	control	Save n for repetition of format code which follows.
nP	scaling factor	control	Save n for use by F, E, and D conversions.
Tn	column reset	control	Reset current position within record to nth column or byte.
nX	skip or blank	control	Skip n characters of an input record or insert n blanks in an output record.
'text' or nH	literal data	control	Move n characters from an input record to the FORMAT statement, or n characters from the FORMAT statement to an output record.
Fw.d Ew.d Dw.d Iw Aw Gw.d Lw Zw	F - conversion E - conversion D - conversion I - conversion A - conversion G - conversion L - conversion Z - conversion	conversion conversion conversion conversion conversion conversion conversion conversion	Exit to the load module to return control to entries FIOLF or FIOAF in IHCFCVTH. Using information passed to the I/O list section, the address and length of the current list item are obtained and passed to the proper conversion routine together with the current position in the I/O buffer, the scale factor, and the values of w and d. Upon return from the conversion routine the current field count is tested. If it is greater than 1, another exit is made to the load module to obtain the address of the next list item.
)	group end	control	Test group count. If greater than 1, repeat format codes in group; otherwise continue to process FORMAT statement from current position.
/	record end	control	Input or output one record via I/O Interface and READ/WRITE macro instruction.
	end of statement	control	If no I/O list items remain to be transmitted, return control to the load module to link to the closing section; if list items remain, input or output one record using I/O interface and READ/WRITE macro instruction. Repeat format codes from last parenthesis.

The selected conversion routine obtains data from an input buffer and converts the data to the form dictated by the conversion code. The converted data is then moved into the main storage address assigned to the list item.

In general, after a conversion routine has processed a list item, the I/O list section determines if that routine can be applied to the next list item or array element (if an array is being processed). The I/O list section examines a field count that indicates the number of times a particular conversion code is to be applied to successive list items or successive elements of an array.

If the conversion code is to be repeated and if the previous list item was a variable, the I/O list section returns control to the load module. The load module again branches to the I/O list section and passes, as an argument, the main storage address assigned to the next list item.

The conversion routine that processed the previous list item is then given control. This procedure is repeated until either the field count is exhausted or the input data for the READ statement is exhausted.

If the conversion code is to be repeated and if an array is being processed, the I/O list section 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 the input data for the READ statement is exhausted.

If the conversion code is not to be repeated, control is passed to the scan portion of IHCFCOMH to continue the scan of the FORMAT statement. 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 statement 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 load module if the previous list item was a variable. The load module again branches to the I/O list section 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, the I/O list section 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. Subsequent I/O list processing for a READ requiring a format proceeds at the point where the field count is examined.

If the scan portion encounters the end of the FORMAT statement and if all the list items are satisfied, control returns to the next sequential instruction within the load module. This instruction (part of the calling sequence to IHCFCOMH) branches to the closing section. If all the list items are not satisfied, control is passed to the I/O interface to read (via the READ macro instruction) the next input record. The conversion codes starting from the last left parenthesis are then repeated for the remaining list items.

If the operation is a write requiring a format, the I/O list section processing is similar to that for a read requiring a format. 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 then transferred to an output buffer. If all the list items have not been converted and transferred before the end-of-the FORMAT statement is encountered, control is passed to the I/O interface. The I/O interface writes (via the WRITE macro instruction) the contents of the current output buffer onto the output data set. The conversion codes starting from the last left parenthesis are then repeated for the remaining list items.

Read/Write Not Requiring a Format: In processing a list item for a read not requiring a format, the I/O list section must know the main storage address assigned to the list item and the size of the list item. Their values are passed, as arguments, via the calling sequence to the I/O list section. The list item may be either a variable or an array. In either case, the number of bytes specified by the size of the list item is moved from the input buffer to the main storage address assigned to the list item. The I/O list section then returns control to the load module. The load module again branches to the I/O list section and passes, as arguments, the main storage address assigned to the next list item and the size of the list item. The I/O list section moves the number of bytes specified by the size of the list item into the main storage address assigned to this list item. This procedure is repeated either until all the list items are satisfied or until the input data is exhausted. Control is then returned to the load module.

If the operation is a write not requiring a format, the I/O list section processing is similar to that described for a read not requiring a format. The main difference is that the data is obtained from the main storage addresses assigned to the list items and is then moved to an output buffer. In addition, the segment length (i.e., the number of bytes in the record segment) and a code indicating the position of this segment relative to other segments, if any, of the logical record are inserted in the segment control word.

CLOSING SECTION: The compiler generates a calling sequence to one of two entry points in the closing section of IHCFCOMH each time it encounters the end of a READ or WRITE statement in the FORTRAN source module. The entry points correspond to the operations of read and write, requiring or not requiring a format.

Read/Write Requiring a Format: If the operation is a read requiring a format, the closing section simply returns control to the load module to continue load module execution. If the operation is a write requiring a format, the closing section branches to the I/O interface. The I/O interface writes (via the WRITE macro instruction) the contents of the current I/O buffer (the final record) onto the output data set. The I/O interface then returns control to the closing section. The closing section, in turn, returns control to the load module to continue load module execution.

Read/Write Not Requiring a Format: If the operation is a read not requiring a format, the closing section branches to the I/O interface. The I/O interface reads (via the READ macro instruction) successive records until the end of the logical record being read is encountered. (A FORTRAN logical record consists of all the records necessary to contain the I/O list items for a WRITE statement not requiring a format.) When the I/O interface recognizes the end-of-logical- record indicator, control is returned to the closing section. The closing section, in turn, returns control to the load module to continue load module execution.

If the operation is a write not requiring a format, the closing section inserts: (1) the segment length (i.e., the number of bytes in the record segment) and a code indicating that this segment is either the last or the only segment of the logical record into the segment control word of the I/O buffer to be written, and (2) an end-of-logical-record indicator into the last record of the I/O buffer being written. The closing section then branches to the I/O interface. The I/O interface writes (via the WRITE macro instruction) the contents of this I/O buffer onto the output data set. The I/O interface then returns control to the closing section. The closing section, in turn, returns control to the load module to continue load module execution.

Examples of IHCFCOMH READ/WRITE Statement Processing

The following examples illustrate the opening section, I/O list section, and closing section processing performed by IHCFCOMH for sequential access READ and WRITE statements, requiring or not requiring a format.

Note: IHCFCOMH processing for the direct access READ, FIND, and WRITE statements is essentially the same as that described for the sequential access READ and WRITE statements. The main difference is that for direct access statements, IHCFCOMH branches to the direct access I/O interface (IHCDIOSH) instead of to the sequential access I/O interface (IHCFIOSH).

READ REQUIRING A FORMAT: The processing performed by IHCFCOMH for the following READ statement and FORMAT statement is illustrated in Table 32.

```
READ (1,2) A,B,C
2 FORMAT (3F12.6)
```

Table 32. IHCFCOMH Processing for a READ Requiring a Format

Opening Section	1. Receives control from load module and branches to IHC-FIOSH to initialize data set for reading. 2. Passes control to scan portion of IHCFCOMH. 3. Returns control to load module.
I/O List Section	1. Receives control from load module, converts input data for A using IHCFCVTH, and moves converted data to A. 2. Returns control to load module. 3. Receives control from load module, converts input data for B, and moves converted data to B. 4. Returns control to load module. 5. Receives control from load module, converts input data for C, and moves converted data to C. 6. Returns control to load module.
Closing Section	1. Receives control from load module and closes out I/O operation. 2. Returns control to load module to continue load module execution.

WRITE REQUIRING A FORMAT: The processing performed by IHCFCOMH for the following WRITE statement and FORMAT statement is illustrated in Table 33.

```
WRITE (3,2) (D(I),I=1,3)
2 FORMAT (3F12.6)
```

Table 33. IHCFCOMH Processing for a WRITE Requiring a Format

Opening Section	1. Receives control from load module and branches to IHC-FIOSH to initialize data set for writing. 2. Passes control to scan portion of IHCFCOMH. 3. Returns control to load module.
I/O List Section	1. Receives control from load module, converts D(1), and moves D(1) to output buffer. 2. Returns control to load module. 3. Receives control from load module, converts D(2), and moves D(2) to output buffer. 4. Returns control to load module. 5. Receives control from load module, converts D(3), and moves D(3) to output buffer. 6. Returns control to load module.
Closing Section	1. Receives control from load module and branches to IHC-FIOSH to write contents of output buffer. 2. Returns control to load module to continue load module execution.

READ NOT REQUIRING A FORMAT: The processing performed by IHCFCOMH for the following READ statement is illustrated in Table 34.

READ (5) X,Y,Z

Table 34. IHCFCOMH Processing for a READ Not Requiring a Format

Opening Section	1. Receives control from load module and branches to IHCFIOSH to initialize data set for reading. 2. Returns control to load module.
I/O List Section	1. Receives control from load module and moves input data to X. 2. Returns control to load module. 3. Receives control from load module and moves input data to Y. 4. Returns control to load module. 5. Receives control from load module and moves input data to Z. 6. Returns control to load module.
Closing Section	1. Receives control from load module and branches to IHCFIOSH to read successive records until the end-of-logical-record indicator is encountered. 2. Returns control to load module to continue load module execution.

WRITE NOT REQUIRING A FORMAT: The processing performed by IHCFCOMH for the following WRITE statement is illustrated in Table 35.

WRITE (6) (W(J),J=1,10)

•Table 35. IHCFCOMH Processing for a WRITE Not Requiring a Format

Opening Section	1. Receives control from load module and branches to IHCFIOSH to initialize data for writing. 2. Returns control to load module.
I/O List Section	1. Receives control from load module and moves W(1) to output buffer. 2. Returns control to load module. 3. Receives control from load module and moves W(2) to output buffer. 4. Returns control to load module. . . . 5. Receives control from load module and moves W(10) to output buffer. 6. Returns control to load module.
Closing Section	1. Receives control from load module, inserts control information, and branches to IHCFIOSH to write contents of output buffer. 2. Returns control to load module to continue load module execution.

READ/WRITE Statement Using NAMELIST

Included in the calling sequence to IHCNAMEL[1] generated by the compiler when it detects a READ or WRITE using a NAMELIST is a pointer to the object-time namelist dictionary associated with the READ or WRITE. This dictionary contains the names and addresses of the variables and arrays into which data is to be read or from which data is to be written. The dictionary also contains the information needed to select the conversion routine that is to convert the data to be placed into the variables or arrays, or to be taken from the variables and arrays.

READ USING NAMELIST: The data set containing the data to be input to the variables or arrays is initialized and successive records are read until the one containing the namelist name corresponding to that in the namelist dictionary is encountered. The next record is then read and processed.

The record is scanned and the first name is obtained. The name is compared to the variable and array names in the namelist dictionary. If the name does not agree, an error is signaled and load module execution is terminated. If the name is in the dictionary, processing of the matched variable or array is initiated.

Each initialization constant assigned to the variable or an array element is obtained from the input record. (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 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.

The process is repeated for the second and subsequent names in the input record. When an entire record has been processed, the next 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.

----

[1]IHCNAMEL is included in the load module only if reads and writes using NAMELISTs appear in the compiled program. Calls are made directly to FRDNL# (for READ) or to FWRNL# (for WRITE).

WRITE USING NAMELIST: The data set upon which the variables and arrays are to be written is initialized. The namelist name is obtained from the namelist dictionary associated with the WRITE, moved to an I/O buffer, and written. The processing of the variables and arrays is then initiated.

The first variable or array name in the dictionary is moved to an I/O buffer followed by an equal sign. The appropriate conversion routine is selected according to the type of the 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 I/C 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 I/O 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.

I/O Device Manipulation Routines

The I/O device manipulation routines of IHCFCOMH implement the BACKSPACE, REWIND, and END FILE source statements. These routines receive control from within the load module via calling sequences that are generated by the compiler when these statements are encountered.

Note: The I/O device manipulation routines apply only to sequential access I/O devices (e.g., tape units). BACKSPACE, REWIND, and ENDFILE requests for direct access data sets are ignored.

The implementation of REWIND and END FILE statements is straightforward. The I/O device manipulation routines submit the appropriate control request to IHCFIOSH, the I/O interface module. After the request is executed, control is returned to the calling routine within the load module.

The BACKSPACE statement is processed in a similar fashion. However, before control is returned to the calling routine, it is determined whether the record backspaced over is an element of a data set that does not require a format. If the record is an element of such a data set, that record is read into an I/O buffer and the segment control word is examined. If it indicates that the record is the first or only segment of the logical record, a backspace control request is issued and control is returned to the calling routine. If the segment control word indicates that this is the last or an intermediate segment, two backspace control requests are issued to backspace to the beginning of the preceding record segment. This record is then read in and its segment control word examined. If it is still not the first segment, two more backspace control requests are issued. This process continues until the first segment is read. Then a backspace control request is issued and control is returned to the calling routine. If the record is not an element of such a data set, control is returned directly to the calling routine.

Write-to-Operator Routines

The write-to-operator routines of IHCFCOMH implement the STOP and PAUSE source statements. These routines receive control from within the load module via calling sequences generated by the compiler upon recognition of the STOP and PAUSE statements.

STOP: A write-to-operator (WTO) macro instruction is issued to display the message associated with the STOP statement on the console. Load module execution is then terminated by passing control to the program termination routine of IHCFCOMH.

PAUSE: A write-to-operator-with-reply (WTOR) macro instruction is issued to display the message associated with the PAUSE statement on the console and to enable the operator's reply to be transmitted. A WAIT macro instruction is then issued to determine when the operator's reply has been transmitted. After the reply has been received, control is returned to the calling routine within the load module.

Utility Routines

The utility routines of IHCFCOMH perform the following functions:

- Process object-time error messages.
- Process arithmetic-type program interruptions.
- Process specification interruptions.
- Terminate load module execution.

PROCESSING OF ERROR MESSAGES: The error message processing routine (IBFERR) receives control from various FORTRAN library subprograms when they detect terminal object-time errors.

Error message processing consists of initializing the data set upon which the message is to be written and of writing the message and a diagnostic traceback. Control is then passed to the routine for terminating load module execution.

PROCESSING OF INTERRUPTIONS: The interrupt routine (IBFINT) of IHCFCOMH initially receives control from within the load module via a compiler-generated calling sequence. The call is placed at the start of the executable coding of the load module so that the interrupt routine can set up the program interrupt mask. Subsequent entries into the interrupt routine are made through specification or arithmetic-type interruptions.

The interrupt routine sets up the program interrupt mask by means of a SPIE macro instruction. This instruction specifies the type of interruptions that are to cause control to be passed to the interrupt routine, and the location within the routine to which control is to be passed if the specified interruptions occur. After the mask has been set, control is returned to the calling routine within the load module.

In processing an interruption, the first step taken by the interrupt routine is to determine its type.

A. Arithmetic Interruptions: If exponential overflow or underflow has occurred, the appropriate indicators, which are referred to by OVERFL (a library subprogram), are set. If any type of divide check caused the interruption, the indicator referred to by DVCHK (also a library subprogram) is set.

Regardless of the type of interruption that caused control to be given to the interrupt routine, the old program PSW is written out for diagnostic purposes.

After the interruption has been processed, control is returned to the interrupted routine at the point of interruption.

B. Specification Interruptions: If an interrupt is caused by a specification exception and the boundary alignment option was specified by the user during system generation, the boundary adjustment routine (IHCADJST) is loaded from the link library (SYS1.LINKLIB).

This routine determines whether or not the interruption was caused by an instruction that referred to improperly aligned data. If not, the routine causes abnormal termination of the load module. If so, the routine:

1. Causes message IHC210I, which contains the main program PSW, to be generated.

2. Moves the misaligned data to a properly aligned boundary.

3. Reexecutes the instruction that refers to the data.

If no interruption occurs when the instruction is reexecuted, the data is moved back to its original location. If there is a new condition code, it is placed in the PSW of the Program Interruption Element (PIE). The boundary adjustment routine then returns control to the control program, which loads the PSW of the PIE to effect a return to the interrupted program.

If a divide check, exponential overflow or underflow interruption occurs when the instruction is reexecuted, the interruption will be handled as described under "Arithmetic Interruptions."

If a data, protection, or addressing interruption occurs when the instruction is reexecuted, the boundary adjustment routine generates the message IHC210I. The PSW information in this message gives the cause of the interruption and the location of the instruction in the main program that caused the interruption. Then, since processing cannot continue, the routine issues a SPIE macro instruction to remove specification interruptions from those interruptions handled by this routine and reexecutes the instruction. This causes abnormal termination of the load module because of the original specification error.

PROGRAM TERMINATION: The load module termination routine (IBEXIT) of IHCFCOMH receives control from various library subprograms (e.g., DUMP and EXIT) and from other IHCFCOMH routines (e.g., the routine that processes the STOP statement).

This routine terminates execution of the load module by the following means:

- Calling the appropriate I/O interface (s) to check (via the CHECK macro instruction) outstanding write requests.

- Issuing a SPIE macro instruction with no parameters indicating that the FORTRAN object module no longer desires to give special treatment to program interruptions and does not want maskable interruptions to occur.

- Returning to the operating system supervisor.

CONVERSION ROUTINES (IHCFCVTH)

The conversion routines (refer to Table 37) either convert data to be placed into I/O list items or convert data to be taken from I/O list items.

These routines receive control either from the I/O list section of IHCFCOMH during its processing of list items for READ/WRITE statements requiring a format, from the routines that process READ/WRITE statements using a NAMELIST, or from the DUMP and PDUMP subprograms.

Each conversion routine is associated with a conversion type format code and/or a type. If an I/O list item for READ/WRITE statement requiring a format is being processed, the conversion routine is selected according to the conversion type format code which is to be applied to the list item. If a list item for a READ/WRITE using a NAMELIST is being processed, the conversion routine is selected according to the type of the list item.

If a READ statement is being implemented, the conversion routine obtains data from the I/O buffer, converts it according to its associated conversion type format code or type, and enters the converted data into the list item. The process is reversed if a WRITE statement is being implemented.

For the DUMP and PDUMP subprograms, the format code parameter passed to them determines the selection of the output conversion routine to be used to place the output in the desired form.

IHCFIOSH

IHCFIOSH, the object-time FORTRAN sequential access input/output data management interface, receives I/O requests from IHCFCOMH and submits them to the appropriate BSAM (basic sequential access method) routines and/or open and close routines for execution.

Chart 26 illustrates the overall logic and the relationship among the routines of IHCFIOSH. Table 38, the IHCFIOSH routine directory, lists the routines used in IHCFIOSH and their functions.

BLOCKS AND TABLES USED

IHCFIOSH uses the following blocks and table during its processing of sequential access input/output requests: (1) unit blocks, and (2) unit assignment table. The unit blocks are used to indicate I/O activity for each unit number (i.e., data set reference number) and to indicate the type of operation requested. In addition, the unit blocks contain skeletons of the data event control blocks (DECB) and the data control blocks (DCB) that are required for I/O operations. The unit assignment table is used as an index to the unit blocks.

Unit Blocks

The first reference to each unit number (data set reference number) by an input/output operation within the FORTRAN load module causes IHCFIOSH to construct a unit block for each unit number. The main storage for the unit blocks is obtained by IHCFIOSH via the GETMAIN macro instruction. The addresses of the unit blocks are placed in the unit assignment table as the unit blocks are constructed. All subsequent references to the unit numbers are then made through the unit assignment table. Figure 57 illustrates the format of a unit block for a unit that is defined as a sequential access data set.

ABYTE	BBYTE	CBYTE	LIVECNT	Housekeeping Section
Address of Buffer 1				
Address of Buffer 2				
Current buffer pointer				
Record offset				
DECB skeleton section				
DCB skeleton section				

Figure 57. Format of a Unit Block for a Sequential Access Data Set

Each unit block is divided into three sections: a housekeeping section, a DECB skeleton section, and a DCB skeleton section.

HOUSEKEEPING SECTION: The housekeeping section is maintained by IHCFIOSH. The information contained in it is used to indicate data set type, to keep track of I/O buffer locations, and to keep track of addresses internal to the I/O buffers to enable the processing of blocked records. The fields of this section are:

- ABYTE. This field, containing the data set type passed to IHCFIOSH by IHCF-COMH, can be set to one of the following:

  F0 - Input data set requiring a format.
  FF - Output data set requiring a format.
  00 - Input data set not requiring a format.
  0F - Output data set not requiring a format.

- BBYTE. This field contains bits that are set and examined by IHCFIOSH during its processing. The bits and their meanings are as follows:

  Bit on

  0 - exit to IHCFCOMH on I/O error
  1 - I/O 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 IHCFIOSH. The bits and their meanings are as follows:

  Bit on

  0 - data control block opened
  1 - data control block not TCLOSEd
  2 - data control block not previously opened
  3 - buffer pool attached
  4 - data set not previously rewound
  5 - data set not previously backspaced
  6 - concatenation occurring -- reissue READ
  7 - data set is DUMMY

- LIVECNT. This field indicates whether any I/O operation performed for this 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 all previous read and write operations for this data set have been checked.)

- Address of Buffer 1 and Address of Buffer 2. These fields contain pointers to the two I/O buffers obtained during the opening of the data control block for this data set.

- Current Buffer Pointer. This field contains a pointer to the I/O buffer currently being used.

- Record Offset. This field contains a pointer to the current logical record within the current buffer.

DECB SKELETON SECTION: The DECB (data event control block) skeleton section is a block of main storage within the unit block. It is of the same form as the DECB constructed by the control program for an L form of an S-type READ or WRITE macro instruction (refer to the publication IBM System/360 Operating System: Supervisor and Data Management Macro Instructions). The various fields of the DECB skeleton are filled in by IHCFIOSH; the completed block is referred to when IHCFIOSH issues a read/write request to BSAM. The read/write field is filled in at open time. For each I/O operation, IHCFIOSH supplies IHCFCOMH with: (1) an indication of the type of operation (read or write), and (2) the length of and a pointer to the I/O buffer to be used for the operation.

DCB SKELETON SECTION: The DCB (data control block) skeleton section is a block of main storage within the unit block. It is of the same form as the DCB constructed by the control program for a DCB macro instruction under BSAM (refer to the publication IBM System/360 Operating System: Supervisor and Data Management Macro Instructions). The various fields of the DCB skeleton are filled in by the control program when the DCB for the data set is opened (refer to the publication IBM System/360 Operating System: Concepts and Facilities). (Standard default values may also be inserted in the DCB skeleton by IHCFIOSH. Refer to "Unit Assignment Table" for a discussion of when default values are inserted into the DCB skeleton.)

Unit Assignment Table

The unit assignment table (IHCUATBL) resides on the FORTRAN system library (SYS1.FORTLIB). Its size depends on the maximum number of units that can be referred to during execution of any FORTRAN load module. This number (≥ 99) is specified by the user during the system generation process via the FORTLIB macro instruction.

The unit assignment table is designed to be used by both IHCFIOSH and IHCDIOSH. It is included once, by the linkage editor, in the FORTRAN load module as a result of an external reference to it within IHCFIOSH and/or IHCDIOSH.

The unit assignment table contains a 16 byte entry for each of the unit numbers that can be referred to by the user. These entries differ in format depending on whether the unit has been defined as a sequential access or a direct access data set.

Figure 58 illustrates the format of the unit assignment table.

Unit number (DSRN) being used for current operation			[1]n x 16	4 bytes
ERRMSG DSRN[2]	READ DSRN[3]	PRINT DSRN[4]	PUNCH DSRN[5]	4 bytes
UBLOCK01 field				4 bytes
DSRN01 default values				8 bytes
LIST01 field				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 UBLOCKn 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]The default values for the various unit numbers are specified by the user and are assembled into the unit assignment table entries during the system generation process. The default values are used only by IHCFIOSH; they are ignored by IHCDIOSH.
[8]If the unit is defined as a direct access data set, the LISTn 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. Unit Assignment Table Format

Because IHCFIOSH deals only with sequential access data sets, the remainder of the discussion on the unit assignment table is devoted to unit assignment table entries for sequential access data sets. If IHCFIOSH encounters a reference to a direct access data set, it is considered as an error, and control is passed to the load module termination routine of IHCFCOMH.

The pointers to the unit blocks created for sequential data sets are inserted into the unit assignment table entries by IHC-FIOSH when the unit blocks are constructed.

Note: Default values are standard values that IHCFIOSH inserts into the appropriate fields (e.g., BUFNO) of the DCB skeleton section of the unit blocks if the user either:

- Causes the load module to be executed via a cataloged procedure, or
- Fails, in stating his own procedure for execution, to include in the DCB parameter of his DD statements those sub-parameters (e.g., BUFNO) he is permitted to include (refer to the publication IBM System/360 Operating System: FORTRAN IV (H) Programmer's Guide).

Control is returned to IHCFIOSH during data control block opening so that it can determine if the user has included the sub-parameters in the DCB parameter of his DD statements. IHCFIOSH examines the DCB skeleton fields corresponding to user-permitted subparameters, and upon encountering a null field (indicating that the user has not specified the subparameter), inserts the standard value (i.e., the default value) for the subparameter into the DCB skeleton. (If the user has included these subparameters in his DD statement, the control program routine performing data control block opening inserts the subparameter values, before giving control to IHCFIOSH, into the DCB skeleton fields reserved for those values.)

BUFFERING

All input/output operations are double buffered. (The double buffering scheme can be overridden by the user if he specifies in a DD statement: BUFNO=1.) This implies that during data control block opening, two buffers will be obtained. The addresses of these buffers are given alternately to IHCFCOMH as pointers to:

- Buffers to be filled (in the case of output).
- Information that has been read in and is to be processed (in the case of input).

COMMUNICATION WITH THE CONTROL PROGRAM

In requesting services of the control program, IHCFIOSH uses L and E forms of S-type macro instructions (refer to the publication IBM System/360 Operating System: Supervisor and Data Management Macro Instructions).

OPERATION

The processing of IHCFIOSH is divided into five sections: initialization, read, write, device manipulation, and closing. When called by IHCFCOMH, a section of IHC-FIOSH performs its function and then returns control to IHCFCOMH.

Initialization

The initialization action taken by IHC-FIOSH depends upon the nature of the previous I/O operation requested for the data set. The previous operation possibilities are:

- No previous operation.
- Previous operation read or write.
- Previous operation backspace.
- Previous operation write end-of-data set.
- Previous operation rewind.

NO PREVIOUS OPERATION: If no previous operation has been performed on the unit specified in the I/O request, the initialization section generates a unit block for the unit number. The data set to be created is then opened (if the current operation is not rewind or backspace) via the OPEN macro instruction. The addresses of the I/O buffers, which are obtained during the opening process and placed into the DCB skeleton, are placed into the appropriate fields of the housekeeping section of the unit block. The DECB skeleton is then set to reflect the nature of the operation (read or write), the format of the records to be read or written, and the address of the I/O buffer to be used in the operation.

If the requested operation is a write, a pointer to the buffer position, at which IHCFCOMH is to place the record to be written, and the block size or logical record length (to accommodate blocked logical records) are placed into registers, and control is returned to IHCFCOMH.

If the requested operation is a read, a record is read, via a READ macro instruction, into the I/O buffer, and the operation is checked for completion via the CHECK macro instruction. A pointer to the location of the record within the buffer, along with the number of bytes read or the logical record length, are placed into registers, and control is returned to IHCFCOMH.

Note: During the opening process, control is returned to the IHCDCBXE routine in IHCFIOSH. This routine determines if the data set being opened is a 1403 printer. If it is, the RECFM field in the DCB for the data set is altered to machine carriage control (FM). In addition, a pointer to the unit block generated for the printer, and the physical address of the printer are placed into a control block area (CTLBLK) for the printer within IHCFIOSH. CTLBLK also contains a third print buffer. This buffer is used in conjunction with the two buffers already obtained for the printer.

Figure 59 illustrates the format of CTLBLK.

CTLBLK	a(BUF 3)		4 bytes
	a(unit block)		4 bytes
	a(printer)	record length	4 bytes
	[1] FT00		4 bytes
	[1] F001		4 bytes
BUF3	third print buffer		144 bytes

[1]Used in the task input/output table (TIOT) search.

Figure 59. CTLBLK Format

PREVIOUS OPERATION READ OR WRITE: If the previous operation performed on the unit specified in the present I/O request was either a read or write, the initialization section determines the nature of the present I/O request. If it is a write, a pointer to the buffer position, at which IHCFCOMH is to place the record to be written, and the block size or logical record length are placed into registers, and control is returned to IHCFCOMH.

If the operation to be performed is a read, a pointer to the buffer location of the record to be processed, along with the number of bytes read or logical record length, are placed into registers, and control is returned to IHCFCOMH.

PREVIOUS OPERATION BACKSPACE: If the previous operation performed on the unit specified in the present I/O request was a backspace, the initialization section determines the type of the present operation (read or write) and modifies the DECB skeleton, if necessary, to reflect the operation type. (If the operation type is the same as that of the operation that preceded the backspace request, the DECB skeleton need not be modified.) Subsequent processing steps are the same as those described for "No Previous Operation," starting at the point after the DECB skeleton is set to reflect operation type.

PREVIOUS OPERATION WRITE END-OF-DATA SET: If the previous operation performed on the unit specified in the present I/O request was a write end-of-data set, a new data set using the same unit number is to be created. In this case, the initialization section closes the data set. Then, in order to establish a correspondence between the new data set and the DD statement describing that data set, IHCFIOSH increments the unit sequence number of the ddname. (The ddname is placed into the appropriate field of the DCB skeleton prior to the opening of the initial data set associated with the unit number.) During the opening of the data set, the ddname will be used to merge with the appropriate DD statement. The data set is then opened. Subsequent processing steps are the same as those described for "No Previous Operation," starting at the point after the data set is opened.

PREVIOUS OPERATION REWIND: If the previous operation performed on the unit specified in the present I/O request was a rewind, the ddname is initialized (set to FTxxF001) in order to establish a correspondence between the initial data set associated with the unit number and the DD statement describing that data set. The data set is then opened. Subsequent processing steps are the same as those described for "No Previous Operation," starting at the point after the data set is opened.

Read

The read section of IHCFIOSH performs two functions: (1) reads physical records into the buffers obtained during data set opening, and (2) makes the contents of these buffers available to IHCFCOMH for processing.

If the records being processed are blocked, the read section does not read a physical record each time it is given control. IHCFIOSH only reads a physical record when all of the logical records of the blocked record under consideration have been processed by IHCFCOMH. However, if the records being processed are either unblocked or of U-format, the read section of IHCFIOSH issues a READ macro instruction each time it receives control.

The reading of records by this section is overlapped. That is, while the contents of one buffer are being processed, a physical record is being read into the other buffer. When the contents of one buffer have been processed, the read into the other buffer is checked for completion. Upon completion of the read operation, processing of that buffer's contents is initiated. In addition, a read into the second buffer is initiated.

Each time the read section is given control it makes the next record available to IHCFCOMH for processing. (In the case of blocked records, the record presented to IHCFCOMH is logical.) The read section of IHCFIOSH places: (1) a pointer to the record's location in the current I/O buffer, and (2) the number of bytes read or logical record length into registers, and then returns control to IHCFCOMH.

Write

The write section of IHCFIOSH performs two functions: (1) writes physical records, and (2) provides IHCFCOMH with buffer space in which to place the records to be written.

If the records being written are blocked, the write section does not write a physical record each time it is given control. IHCFIOSH only writes a physical record when all of the logical records that comprise the blocked record under consideration have been placed into the I/O buffer by IHCFCOMH. However, if the records being written are either unblocked or of U-format, the write section of IHCFIOSH issues a WRITE macro instruction each time it receives control.

The writing of records by this section is overlapped. That is, while IHCFCOMH is filling one buffer, the contents of the other buffer are being written. When an entire buffer has been filled, the write from the other buffer is checked for completion. Upon completion of the write operation, IHCFCOMH starts placing records into that buffer. In addition, a write from the second buffer is initiated.

Each time the write section is given control, it provides IHCFCOMH with buffer space in which to place the record to be written. IHCFIOSH places: (1) a pointer to the location within the current buffer at which IHCFCOMH is to place the record, and (2) the block size or logical record length into registers, and then returns control to IHCFCOMH.

Note: The write section checks to see if the data set being written on is a 1403 printer. If it is, the carriage control character is changed to machine code, and three buffers, instead of the normal two, are used when writing on the printer.

ERROR PROCESSING: If an end-of-data set or an I/O error is encountered during reading or writing, the control program returns control to the location within IHCFIOSH that was specified during data set initialization. In the case of an I/O error, IHCFIOSH sets a switch to indicate that the error has occurred. Control is then returned to the control program. The control program completes its processing and returns control to IHCFIOSH, which interrogates the switch, finds it to be set, and passes control to the I/O error routine of IHCFCOMH.

In the case of an end-of-data set, IHCFIOSH simply passes control to the end-of-data set routine of IHCFCOMH.

Chart 27 illustrates the execution-time I/O recovery procedure for any I/O errors detected by the I/O supervisor.

Device Manipulation

The device manipulation section of IHCFIOSH processes backspace, rewind, and write end-of-data set requests.

BACKSPACE: IHCFIOSH processes the backspace request by issuing a BSP (physical backspace) macro instruction. It then places the data set type, which indicates the format requirement, into a register and returns control to IHCFCOMH. (IHCFCOMH needs the data set type to determine its subsequent processing.)

REWIND: IHCFIOSH processes the rewind request by issuing a CLOSE macro instruction, using the REREAD option. This option has the same effect as a rewind. Control is then returned to IHCFCOMH.

WRITE END-OF-DATA SET: IHCFIOSH processes this request by issuing a CLOSE macro instruction, type = T. It then frees the I/O buffers by issuing a FREEPOOL macro instruction, and returns control to IHCFCOMH.

Closing

The closing section of IHCFIOSH examines the entries in the unit assignment table to determine which data control blocks are open. In addition, this section ensures that all write operations for a data set are completed before the data control block for that data set is closed. This is done by issuing a CHECK macro instruction for all double-buffered output data sets. Control is then returned to IHCFCOMH.

Note: If a 1403 printer is being used, a write from the last print buffer is issued to insure that the last line of output is written.

IHCDIOSH

IHCDIOSH, the object-time FORTRAN direct access input/output data management interface, receives I/O requests from IHCFCOMH and submits them to the appropriate BDAM (basic direct access method) routines and/or open and close routines for execution. (For the first I/O request involving a nonexistent data set, the appropriate BSAM routines must be executed prior to linking to the BDAM routines. The BSAM routines format and create a new data set consisting of blank records.)

IHCDIOSH receives control from: (1) the initialization section of the FORTRAN load module if a DEFINE FILE statement is included in the source module, and (2) IHCFCOMH whenever a READ, WRITE, or FIND direct access statement is encountered in the load module.

Charts 28 and 29 illustrate the overall logic and the relationship among the routines of IHCDIOSH. Table 39, the IHCDIOSH routine directory, lists the routines used in IHCDIOSH and their functions.

BLOCKS AND TABLE USED

IHCDIOSH uses the following blocks and table during its processing of direct access input/output requests: (1) unit blocks, and (2) unit assignment table. The unit blocks are used to indicate I/O activity for each unit number (i.e., data set reference number) and to indicate the type of operation requested. In addition, each unit block contains skeletons of the data event control blocks (DECB) and the data control block (DCB) that are required for I/O operations. The unit assignment table is used as an index to the unit blocks.

Unit Blocks

The first reference to each unit number (i.e., data set reference number) by a direct access input/output operation within the FORTRAN load module causes IHCDIOSH to construct a unit block for each of the referenced unit numbers. The main storage for the unit blocks is obtained by IHCDIOSH via the GETMAIN macro instruction. The addresses of the unit blocks are inserted into the corresponding unit assignment table entries as the unit blocks are constructed. Subsequent references to the unit numbers are then made through the unit assignment table.

Figure 60 illustrates the format of a unit block for a unit that has been defined as a direct access data set.

ICTYPE	STATUSU	not used	not used	4 bytes
RECNUM				4 bytes
STATUSA	CURBUF			4 bytes
BLKREFA				4 bytes
STATUSB	NXTBUF			4 bytes
BLKREFB				4 bytes
DECBA				28 bytes
DECBB				28 bytes
DCB				104 bytes

Figure 60. Format of a Unit Block for a Direct Access Data Set

The meanings of the various unit block fields are outlined below.

ICTYPE: This field, containing the data set type passed to IHCDIOSH by IHCFCOMH, can be set to one of the following:

F0 - input data set requiring a format

FF - output data set requiring a format

00 - input data set not 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 meanings are as follows:

Bit on

0 - not used

1 - error occurred

2 - two buffers are being used

3 - data control block for data set is open

4-5 10 - U form specified in DEFSINE FILE statement

01 - E form specified in DEFINE FILE statement

11 - L form specified in DEFINE FILE statement

6-7 not used

Note: IHCDICSH 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 are as follows:

Bit on

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 IHCDIOSH 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 IHCDIOSH 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 IHCDIOSH 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 IHCDIOSH 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 IHCDIOSH 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 IHCDIOSH 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 of 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 (refer to the publication IBM System/360 Operating System: Supervisor and Data Management Macro Instructions).

The various fields of the DECBA skeleton are filled in by the file initialization section of IHCDIOSH after the data control block for the data set is opened. The completed DECB is referred to when IHCDIOSH issues a read or a write request to BDAM. For each I/O operation, IHCDIOSH 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 form as the DCB constructed by the control program for a DCB macro instruction under BDAM (refer to the publication IBM System/360 Operating System: Supervisor and Data Management Macro Instructions).

The various fields of the DCB skeleton are filled in by the control program when the DCB for the data set is opened (refer to the publication IBM System/360 Operating System: System Control Blocks).

Unit Assignment Table

The unit assignment table (IHCUATBL) resides on the FORTRAN system library (SYS1.FORTLIB). Its size depends on the maximum number of units that can be referred to during execution of any FORTRAN load module. This number (≥99) is specified by the user during the system generation process via the FORTLIB macro instruction.

The unit assignment table is designed to be used by both IHCFIOSH and IHCDIOSH. It is included once, by the linkage editor, in the FORTRAN load module as a result of an external reference to it within IHCFIOSH and/or IHCDIOSH.

The unit assignment table contains a 16-byte entry for each of the unit numbers that can be referred to by either IHCDIOSH or IHCFIOSH. These entries differ in format depending on whether the unit has been defined as a direct access or as a sequential access data set. Because IHCDIOSH deals only with direct access data sets, only the entry for a direct access unit is shown here. (Refer to the IHCFIOSH section "Table and Blocks Used", for the format of the unit assignment table as a whole.) If IHCDIOSH encounters a reference to a sequential access data set, it is considered as an error, and control is passed to the load module termination routine of IHCFCOMH.

Figure 61 illustrates the unit assignment table entry format for a direct access data set.

Pointer to unit block xx (UBLOCKxx)	4 bytes
Default values for DSRNxx (only applies to sequential access data sets -- not used by IHCDIOSH)	8 bytes
Pointer to parameter listxx (LISTxx)	4 bytes
UBLOCKxx is the unit block generated for unit number xx. DSRNxx is the unit number for the direct access data set (xx≥99). LISTxx is the parameter list that defines the direct access data set associated with unit number xx.	

Figure 61. Unit Assignment Table Entry for a Direct Access Data Set

The pointers to the unit blocks are inserted into the unit assignment table entries by IHCDIOSH when the unit blocks are constructed.

The pointers to the parameter lists are inserted into the unit assignment table entries by IHCDIOSH when IHCDIOSH receives control from the initialization section of the FORTRAN load module being executed.

BUFFERING

All direct access input/output operations are double-buffered. (The double buffering scheme may be overridden by the user if he specifies in his DD statements: BUFNO=1.) This implies that during data control block opening, two buffers will be obtained for each data set. The addresses of these buffers are given alternately to IHCFCOMH as pointers to:

- Buffers to be filled in the case of output.
- Data that has been read in and is to be processed in the case of input.

Each buffer has its own DECB. This increases I/O efficiency by overlapping of I/O operations.

COMMUNICATION WITH THE CONTROL PROGRAM

In requesting services of the control program BSAM and BDAM routines, IHCDIOSH uses L and E forms of S-type macro instructions (refer to the publication IBM System/360 Operating System: Supervisor and Data Management Macro Instructions).

OPERATION

The processing of IHCDIOSH is divided into five sections: file definition, file initialization, read, write, and termination. When a section receives control, it performs its functions and then returns control to the caller (either the FORTRAN load module or IHCFCOMH).

File Definition Section

The file definition section is entered from the FORTRAN load module, via a compiler-generated calling sequence, if a DEFINE FILE statement is included in the FORTRAN source module. The file definition section performs the following functions:

- Checks for the redefinition of each direct access unit number.
- Enters the address of each direct access unit number's parameter list into the appropriate unit assignment table entry.
- Establishes addressability for IHCDIOSH within IHCFCOMH.

Each direct access unit number appearing in a DEFINE FILE statement is checked to see if it has been defined previously. If it has been defined previously, the current definition is ignored. If it has not been defined previously, the address of its parameter list (i.e., the definition of the unit number) is inserted into the proper entry in the unit assignment table. The next unit number if any is then obtained.

When the last unit number has been processed in the above manner, the file definition section stores the address of IHCDIOSH into the FDIOCS field within IHCFCOMH. This enables IHCFCOMH to link to IHCDIOSH when IHCFCOMH encounters a direct access I/O statement. Control is then returned to the FORTRAN load module to continue normal processing.

File Initialization Section

The file initialization section receives control from IHCFCOMH whenever input or output is requested for a direct access data set. The processing performed by the initialization section depends on whether an I/O operation was previously requested for the data set.

NO PREVIOUS OPERATION: If no operation was previously requested for the data set specified in the current I/O request, the file initialization section first constructs a unit block for the data set. (The GETMAIN macro instruction is used to obtain the main storage for the unit block.) The address of the unit block is inserted into the appropriate entry in the unit assignment table.

The file initialization section then reads the JFCB (job file control block) via the RDJFCB macro instruction. The value in the BUFNO field of the JFCB is inserted into the DCB skeleton in the unit block. This value indicates the number of buffers that are obtained for this data set when its data control block is opened. If the BUFNO field is null (i.e., if the user did not include the BUFNO subparameter in the DD statement for this data set), or other than 1 or 2, the file initialization section inserts a value of two into the DCB skeleton.

The file initialization section next examines the JFCBIND2 field in the JFCB to determine if the data set specified in the current I/O request exists. If the JFCBIND2 field indicates that the specified data set does not exist, and if the current request is a write, a new data set is created. (If the current request is a read, an error is indicated and control is returned to IHCFCOMH to terminate load module execution. If the current request is a find, the request is ignored, and control is returned to IHCFCOMH.) If the JFCBIND2 field indicates that the specified data set already exists, a new data set is not created. The file initialization section processing for a data set to be created, and for a data set that already exists is discussed in the following paragraphs.

Data Set to be Created: The data control block for the new data set is first opened for the BSAM, load mode, WRITE macro instruction. The BSAM WRITE macro instruction is used to create a new data set according to the format specified in the parameter list for the data set in a DEFINE FILE statement. The data control block is then closed. Subsequent file initialization section processing after creating the new data set is the same as that described for a data set that already exists (refer to the section "Data Set Already Exists").

Data Set Already Exists: The data control block for the data set is opened for direct access processing by the BDAM routines. After the data control block is opened, the file initialization section fills in various fields in the unit block:

- The number of records in the data set is inserted into the RECNUM field.
- The address of the DECB skeletons (DECBA and DECBB) are inserted into the CURBUF and the NXTBUF fields, respectively.
- The addresses of the I/O buffers obtained during data control block opening are inserted into the appropriate DECB skeletons.
- The address of the BLKREFA and the BLKREFB fields in the unit block are inserted into the appropriate DECB skeletons.

Note: If the user specifies BUFNO=1 in the DD statement for this data set, only one I/O buffer is obtained during data control block opening. In this case, the NXTBUF field, the BLKREFB field, and the DECBB skeleton are not used.

Subsequent file initialization section processing for the case of no previous operation depends upon the nature of the I/O request (find, read, or write). This processing is the same as that described for the case of a previous operation (refer to the section "Previous Operation").

PREVIOUS OPERATION: If an operation was previously requested for the data set specified in the current I/O request, the file initialization section processing depends upon the nature of the current I/O request.

If the current request is either a find or a read, control is passed to the read section.

If the current request is a write, control is passed to the secondary entry in the write section.

Read Section

The read section of IHCDIOSH processes read and find requests. The read section may be entered either from the file initialization section of IHCDIOSH, or from IHCFCOMH. In either case, the processing performed is the same. In processing read and find requests, the read section performs the following functions:

- Reads physical records into the buffer (s) obtained during data control block opening.
- Makes the contents of these buffers available to IHCFCOMH for processing.
- Updates the associated variable that is defined in the DEFINE FILE statement for the data set.

The read section, upon receiving control, first checks to see if the record to be found or read is already in an I/O buffer. Subsequent read section processing depends upon whether the record is in the buffer.

RECORD IN BUFFER: If a record is in the buffer, the read section determines whether the current request is a find or a read.

If the current request is a find, the associated variable for the data set is updated so that it points to the relative position within the direct access data set of the record that is in the buffer. Control is then returned to IHCFCOMH.

If the current request is a read, the read operation that read the record into the buffer is checked for completion. The read section then places the address of the buffer and the size of the buffer into registers for use by IHCFCOMH. The associated variable for the data set is updated so that it points to the relative position within the direct access data set of the record following the record just read. Control is then returned to IHCFCOMH.

RECORD NOT IN BUFFER: If a record is not in the buffer, the read section first obtains the address of the buffer to be used for the current request. The relative record number of the record to be read is then inserted into the appropriate BLKREF field in the unit block (i.e., BLKREFA or BLKREFB). The proper record is then read from the specified data set into the buffer. Subsequent read section processing for the case of a record not in the buffer is the same as that described for a record in the buffer (refer to the section "Record In Buffer").

Note 1: Record retrieval can proceed concurrently with CPU processing only if the user alternates FIND statements with READ statements in his program.

Note 2: If an I/O error occurs during reading, the control program returns control to the synchronous exit routine (SYNADR) within IHCDIOSH. The SYNADR routine sets a switch to indicate that an I/O error has occurred, and then returns control to the control program. The control program completes its processing and returns control to IHCDIOSH. IHCDIOSH interrogates the switch, finds it to be set, and passes control to the I/O error routine of IHCFCOMH.

Write Section

The write section of IHCDIOSH processes write requests. The write section may be entered either from the file initialization section of IHCDIOSH, or from IHCFCOMH. The processing performed by the write section depends upon where it is entered from.

PROCESSING IF ENTERED FROM FILE INITIALIZATION SECTION: If the write section is entered from the file initialization section of IHCDIOSH, no writing is performed. The write section only provides IHCFCOMH with buffer space in which to place the record to be written. The relative record number of the record to be written is inserted into the appropriate BLKREF field (i.e., BLKREFA or BLKREFB). (The record is written the next time the write section is entered.) For a formatted write, the buffer is filled with blanks. For an unformatted write, the buffer is filled with zeros. The write section then places the address of the buffer and the size of the buffer into registers for use by IHCFCOMH. Control is then returned to IHCFCOMH.

PROCESSING IF ENTERED FROM IHCFCOMH: Each time the write section is entered from IHCFCOMH, it writes the contents of the buffer onto the specified data set. Subsequent write section processing for entrances from IHCFCOMH is the same as that described for entrances from the file initialization section of IHCDIOSH (refer to "Processing If Entered From File Initialization Section"). In addition, the associated variable is modified prior to returning to IHCFCOMH. The associated variable for the data set is updated so that it points to the relative position within the direct access data set of the record following the record just written.

Note 1: The writing of physical records by this section is overlapped. That is, while IHCFCOMH is filling buffer A, buffer B is being written onto the output data set. When buffer A has been filled, the write from buffer B is checked for completion. Upon completion of the write operation, IHCFCOMH starts placing data into buffer B. In addition, a write from buffer A is initiated.

Note 2: If an I/O error occurs during writing, the control program returns control to the synchronous exit routine (SYNADR) within IHCDIOSH. The SYNADR routine sets a switch to indicate that an I/O error has occurred, and then returns control to the control program. The control program completes its processing and returns control to IHCDIOSH. IHCDIOSH interrogates the switch, finds it to be set, and passes control to the I/O error routine of IHCFCOMH.

Termination Section

The termination section of IHCDIOSH receives control from the load module termination routine of IHCFCOMH. The function of this section is to terminate any pending I/O operations involving direct access data sets. The unit blocks associated with the direct access data sets are examined by IHCDIOSH to determine if any I/O is pending. CHECK macro instructions are issued for all pending I/O operations to insure their completion.

The data control blocks for the direct access data sets are closed, and the main storage occupied by the unit blocks is freed via the FREEMAIN macro instruction. Control is then returned to the load module termination routine of IHCFCOMH to complete the termination process.

IHCIBERH

IHCIBERH, a member of the FORTRAN system library (SYS1.FORTLIB), processes object-time source statement errors. IHCIBERH is entered when an internal statement number (ISN) cannot be executed because of a source statement error.

The ISN of the invalid source statement is obtained (from information in the calling sequence) and is then converted to decimal form. IHCIBERH then links to IHCFCOMH to implement the writing of the following error message:

IHC230I - SOURCE ERROR AT ISN
XXXX - EXECUTION FAILED SUBROUTINE (name)

After the error message is written on the user-designated error output data set, IHCIBERH passes control to the IBEXIT routine of IHCFCOMH to terminate execution.

Chart 30 illustrates the overall logic of IHCIBERR.

IHCDBUG

IHCDBUG performs the object-time operations of the Debug Facility statements. All linkages from the load module to IHCDBUG are compiler generated.

Items and Buffer

The following items in IHCDBUG are initialized to zero at load time:

- DSRN - the data set reference number
- TRACFLAG - trace flag
- IOFLAG - input/output in progress flag
- DATATYPE - variable type bits

Whenever information is assembled for output, it is placed in a 70-byte area called DBUFFER. The first character of this area is permanently set to blank, for single spacing.

Operation

The first portion of IHCDBUG, called by entry name DEBUG#, is a transfer table; this table is referred to by the code generated for the Debug Facility statements, and branches to the thirteen section of IHCDBUG. These sections are discussed individually.

TRACE ENTRY: If TRACFLAG is off, this routine exits. Otherwise, the characters 'TRACE' are moved to DBUFFER + 1, the subroutine OUTINT converts the statement label to EBCDIC and places it in DBUFFER, and a branch is made to OUTBUFFR.

SUBTRACE ENTRY: The characters 'SUBTRACE' and the name of the program or subprogram are moved to DBUFFER and a branch to OUTBUFFR is made.

SUBTRACE RETURN ENTRY: The characters 'SUBTRACE *RETURN*' are moved to DBUFFER and a branch to OUTBUFFR takes place.

UNIT ENTRY: The unit number argument is placed in DSRN and the routine exits.

INIT SCALAR ENTRY: The data type is saved, the location of the scalar is computed, subroutine OUTNAME places the name of the scalar in DBUFFER, and a branch is made to OUTITEM.

INIT ARRAY ELEMENT ENTRY: This routine saves the data type, computes the location of the array element, and (via the subroutine OUTNAME) places the name of the array in DBUFFER. It then computes the element number as follows:

element number = ((element location - first array location) / element size) + 1

and places a left parenthesis, the element number (converted to EBCDIC by subroutine OUTINT), and a right parenthesis in DBUFFER following the array name. A branch is then made to OUTITEM.

INIT FULL ARRAY ENTRY: If IOFLAG is on, the character X'FF' is placed in DBUFFER, followed by the address of the argument list, and a branch is made to OUTBUFFR. Otherwise, a call to the INIT ARRAY ELEMENT entry is constructed, and the routine loops through that call until all elements of the array have been processed, when it exits.

SUBSCRIPT CHECK ENTRY: The location of the array element is computed; if it is less than or equal to the maximum array location, the routine exits. If the array element location is outside the bounds of the array, the element number is computed and the characters 'SUBCHK' are placed in DBUFFER. The subroutine OUTNAME then places the name of the array in DBUFFER, OUTINT supplies the EBCDIC code for the element number (which is enclosed in parentheses), and a branch is made to OUTBUFFR.

TRACE ON ENTRY: TRACFLAG is turned on (set to nonzero) and the routine exits.

TRACE OFF ENTRY: TRACFLAG is turned off (set to zero) and the routine exits.

DISPLAY ENTRY: If IOFLAG is on, the characters 'DISPLAY DURING I/O SKIPPED' are moved to DBUFFER and a branch is made to OUTBUFFR. Otherwise, a calling sequence for the NAMELIST write routine is constructed. If DSRN is equal to zero, the unit number for SYSOUT (in IHCUATBL + 6) is used as the unit passed to the NAMELIST write routine. On return from the NAMELIST write, this routine exits.

START I/O ENTRY: The BYTECNT is set to 252 to indicate that the current area is full, the IOFLAG is set to X'80' to indicate that input/output is in progress, the CURBYTLC is set to the address of SAVESTRT (where the location of the first main block will be - refer to the description of ALLOCHAR), and the routine exits.

END I/O ENTRY: The IOFLAG is saved in TEMPFLAG and IOFLAG is reset to zero so that this section may make debug calls which result in output to a device. If no information was saved during the input/output, this routine exits.

The subroutine FREECHAR is used to extract one character at a time from the save area. If an X'FF' is encountered (indicating the output of a full array), the next three bytes give the address of the call to INIT FULL ARRAY entry. A call to the DEBUG INIT FULL ARRAY entry is then constructed and executed. If X'FF' is not encountered, characters are placed in DBUFFER until an X'15' is found, indicating the end of a line. When this code is found, the subroutine OUTPUT is used to write out the line.

If no main storage or insufficient main storage was available for saving information during the input/output, the characters 'SOME DEBUG OUTPUT MISSING' are placed in DBUFFER after all saved information (if any) has been written out. The subroutine OUTPUT is then used to write out the message, and this routine returns to the caller.

Subroutines

The following subroutines are used by the routines in IHCDBUG.

OUTITEM: First, the characters ' = ' are moved to DBUFFER. The routine then loads the data to be output into registers. A branch on type then takes place. For fixed point, the routine OUTINT converts the value to EBCDIC and places it in DBUFFER. A branch to OUTBUFFR then takes place.

For floating values, subroutine OUTFLOAT places the value in DBUFFER. A branch to OUTBUFFR then takes place.

For complex values, two calls to OUTFLOAT are made -- first with the real part, then with the imaginary part. A left parenthesis is placed in DBUFFER before the first call, a comma after the first call, and a right parenthesis after the second call. A branch to OUTBUFFR then takes place.

For logical values, a T is placed in DBUFFER if the value was nonzero; otherwise an F is placed in DBUFFER. A branch to OUTBUFFR then takes place.

OUTNAME: This is a closed subroutine. Up to six characters of the name are placed in DBUFFER. However, the first blank in the name causes the routine to exit.

OUTINT: This is a closed subroutine. If the value (passed in R2) is equal to zero, the character '0' is placed in DBUFFER and the routine exits. If it is less than zero, a minus sign is placed in DBUFFER.

The value is then converted to EBCDIC and placed in DBUFFER with leading zeros suppressed. The routine then exits.

OUTFLOAT: This is a closed subroutine. If the value is zero, the characters '0.0E+00' or '0.0D+00' are placed in DBUFFER, depending upon whether the value is single or double-precision, respectively, and the routine exits. If the values are less than zero, a minus sign is placed in DBUFFER. The floating number is then converted to a string of decimal EBCDIC characters and a power of ten by exactly the same algorithm used in IHCFCUTH (this assures identical results).

Let $x = 8$ for single-precision,

$x = 17$ for double-precision.

If $1 \geq |\text{value}| < 10$, it is output to the DBUFFER in Fx+1.x-n format where n is the integer portion of log |value|.

Otherwise it is output in G x+5.x format. The routine then exits.

OUTBUFFR: If IOFLAG is not set, the routine calls the subroutine CUTPUT and then exits. Otherwise, IOFLAG is set to indicate that debug output during input/output occurred. Then, a call is made to ALLOCHAR for each character in DBUFFER, and finally, a call to ALLOCHAR with X'15' indicating the end of the line. The routine then exits.

ALLOCHAR: This is a closed subroutine. If BYTECNT is equal to 252, indicating the current block is full, a new block of 256 bytes is obtained by a GETMAIN macro. If no storage was available, an X'07', indicating end of core, is placed in the last available byte position, IOFLAG is set to full, and the routine exits. Otherwise, the address of the new block is placed in the last three bytes of the previous block, preceded by X'37' indicating end of block with new block to follow. CURBYTLC is then set to the address of the new block and BYTECNT is set to zero. The character passed as an argument is then placed in the byte pointed to by CURBYTLC, one is added to both CURBYTLC and BYTECNT, and the routine exits.

FREECHAR: This is a closed subroutine. If the current character extracted is X'37', the next three bytes are placed in CURBYTLC and the current block is freed. If the current character is X'07' the block is freed and a branch to the End I/O exit is taken. Otherwise, the current character is passed to the calling routine and CURBYTLC is incremented by 1.

OUTPUT: This is a closed subroutine. If DSRN is zero, the SYSOUT unit number is obtained from IHCUATBL + 6. A call is then made to FIOCS# output initialize, DBUFFER is transferred to the FIOCS# buffer, and a call is made to FIOCS# output. The routine then exits.

IHCTRCH

IHCTRCH, a member of the FORTRAN system library (SYS1.FORTLIB) processes terminal errors detected by FORTRAN library subroutines at object time. IHCTRCH is entered only from the IBFERR routine within IBCF-COMH. IBFERR consists only of a call to IHCTRCH.

IHCTRCH issues the following message:

```
IHCxxxI
TRACEBACK FOLLOWS   ROUTINE   ISN   REG. 14
```

where xxx is the error code (in decimal form) that it obtains from the calling sequence.

If the error occurred in IHCFCOMH, IHCFCVTH, IHCNAMEL, IHCDIOSE, or IHCFIOSH, IHCTRCH sets up an area which can be processed as a standard save area for the first traceback line.

For each traceback line, IHCTRCH gets the name of the called routine, the internal statement number, if any, of the call within the calling routine, and the contents of register 14, in hexadecimal.

After printing each line, IHCTRCH checks whether the called routine was the main FORTRAN routine. If so, it prints the entry point, in hexadecimal, and branches to IBEXIT. If not, it enters a traceback loop-check routine, which builds and checks a table of save area addresses. If the table is full or if a loop is detected, IHCTRCH prints TRACEBACK TERMINATED and then prints the main FORTRAN routine entry point and branches to IBEXIT.

IHCTRCH uses IHCFCVTH to convert to printable hexadecimal format and it uses IHCFIOSH for printing.

Further information about traceback including an example of output is contained in the publication IBM System/360 Operating System: FORTRAN IV (H) Programmer's Guide.

Chart 23. IHCFCOMH Overall Logic and Utility Routines

IBCCM

SEE TABLE 33 FOR A BRIEF DISCUSSION OF EACH ROUTINE OF IHCFCOMH.

A3: LOAD MODULE

THE LOAD MODULE ENTERS IHCFCOMH VIA A COMPILER-GENERATED CALLING SEQUENCE.

B3: DETERMINE REQUEST TYPE

REQUEST TYPE	CHART	MAJOR PROCESSING ROUTINES	SUBROUTINES CALLED
SEQUENTIAL ACCESS AND DIRECT ACCESS READ/WRITE RE-QUIRING A FORMAT	24A2	FRDWF,FWRWF,FIOLF, FIOAF,FENDF	IHCFIOSH (FOR SEQUENTIAL ACCESS), IHCDIOSH (FOR DIRECT ACCESS), AND CONVERSION ROUTINES -- FCVII, FCVIO,FCVEI,FCVEO,FCVGI,FCVGO,FCVDI, FCVDO,FCVLI,FCVLO,FCVZI,FCVZO,FCVFI, FCVFO,FCVAI,FCVAO
SEQUENTIAL ACCESS AND DIRECT ACCESS READ/WRITE NOT REQUIRING A FORMAT	24F2	FRDNF,FWRNF,FICLN, FIOAN,FENDN	IHCFIOSH (FOR SEQUENTIAL ACCESS) IHCDIOSH (FOR DIRECT ACCESS)
READ USING NAMELIST	25F1	FRDNL	IHCFIOSH AND CONV. RTNS - FCVEI,FCVDI,FCVAI,FCVLI, FCVGI,FCVCI,FCVII,FCVFI
WRITE USING NAMELIST	25E5	FWDNL	IHCFIOSH AND CONV. RTNS - FCVEO,FCVDO,FCVAO,FCVLO, FCVGO,FCVCO,FCVIO,FCVFO
DEVICE MANIPULATION	25B3	FBKSP,FRWND, FEOFM	IHCFIOSH
WRITE TO OPERATOR	25G3	FSTOP,FPAUS	NONE
DIRECT ACCESS FIND	24F2	FRDNF,FENDN	IHCDIOSH

UTILITY ROUTINES

IBEXIT

G1: FROM FSTOP OR IBFERR

H1: IBEXIT — CLOSE DATA SETS (TERMINATE EXECUTION)

J1: TO OPERATING SYSTEM

IBFERR

G2: FROM LIBRARY SUBPROGRAMS

H2: IBFERR — PROCESS ERRORS

J2: TO IBEXIT

EXCEPT

G3: FROM IHCFIOSH

H3: EXCEPT — DETERMINE IF END PARAMETER SPECIFIED

J3: TO LOAD MODULE IF SPECIFIED

IF PARAMETER NOT SPECIFIED, EXIT IS TO IBFERR.

IBFINT

G4: FROM LOAD MODULE

H4: IBFINT — PROCESS ARITHMETIC INTERRUPTION

J4: TO LOAD MODULE

FERROR

G5: FROM IHCFIOSH OR IHCDIOSH

H5: FERROR — DETERMINE IF ERR PARAMETER SPECIFIED

J5: TO LOAD MODULE IF SPECIFIED

IF PARAMETER NOT SPECIFIED, EXIT IS TO IBFERR.

Chart 24. Implementation of READ/WRITE/FIND Source Statements

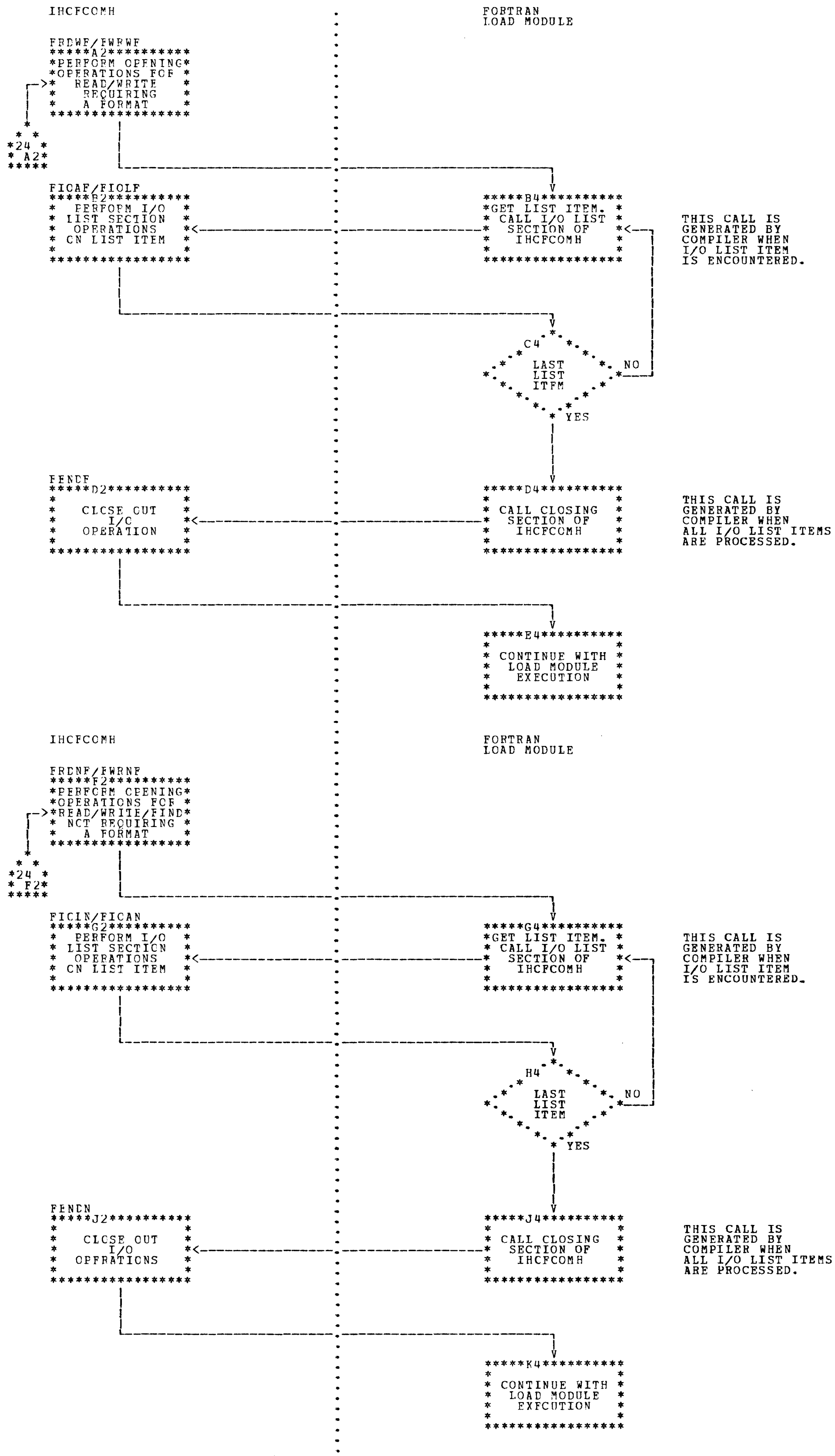


Chart 25. Device Manipulation, Write-to-Operator, and READ/WRITE Using NAMELIST Routines

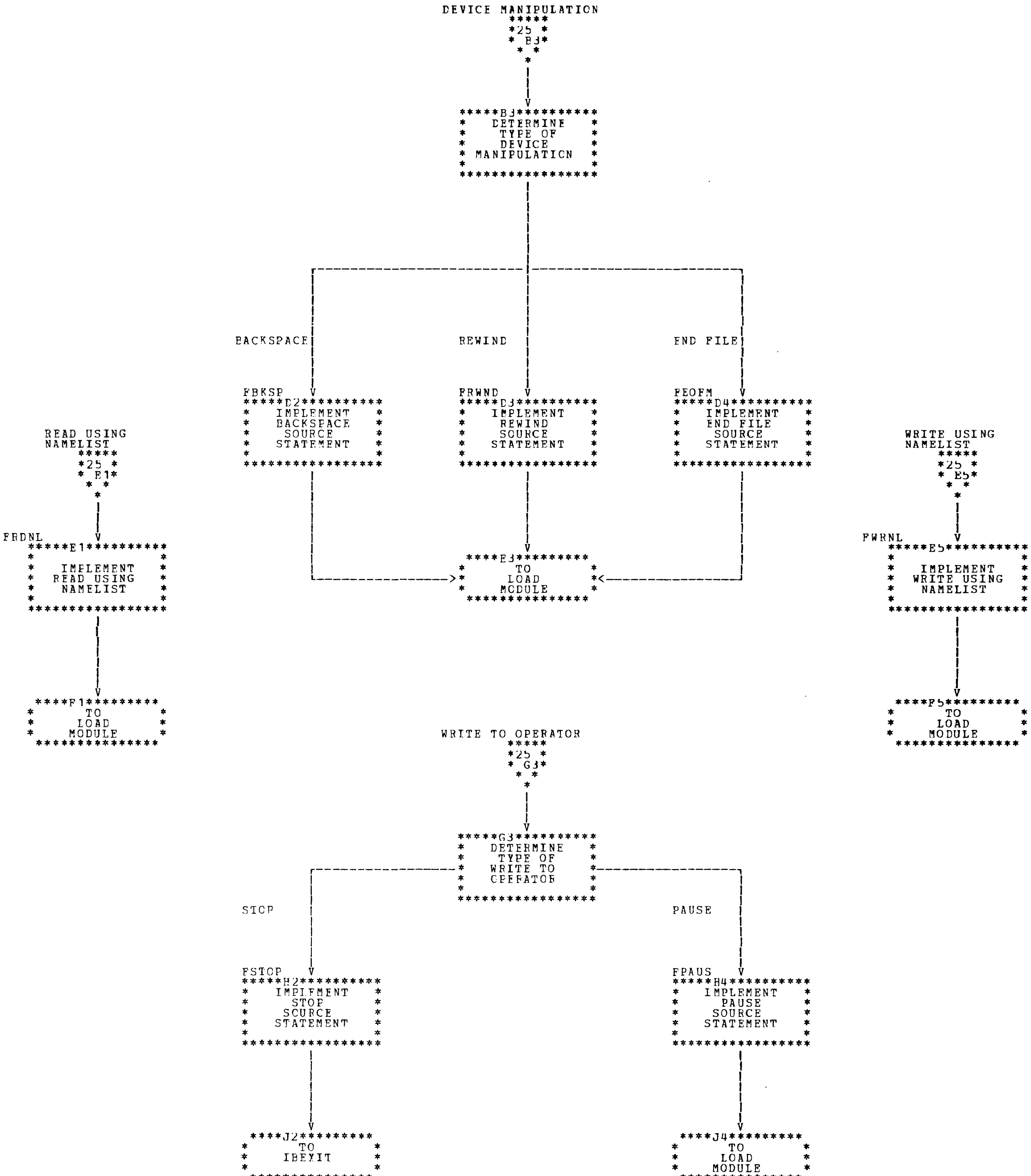


Table 36. IHCFCOMH Subroutine Directory

Subroutine	Function
EXCEPT	Checks for presence of END= parameter, and passes control to the load module if present.
FENDF	Closing section for a READ or WRITE requiring a format.
FENDN	Closing section for a READ or WRITE not requiring a format.
FEOFM	Implements the END FILE source statement.
FERROR	Checks for the presence of the ERR= parameter, and passes control to the load module if present.
FIOAF	I/O list section for list array of a READ or WRITE requiring a format.
FIOAN	I/O list section for list array of a READ or WRITE not requiring a format.
FIOLF	I/O list section for a list variable of a READ or WRITE requiring a format.
FIOLN	I/O list section for a list variable of a READ or WRITE not requiring a format.
FPAUS	Implements the PAUSE source statement.
FRDNF	Opening section of a READ not requiring a format.
FRDWF	Opening section of a READ requiring a format.
FRWND	Implements the REWIND source statement.
FSTOP	Implements the STOP source statement.
FWRNF	Opening section for WRITE not requiring a format.
FWRWF	Opening section for WRITE requiring a format.
IBEXIT	Closes all data sets and terminates execution.
IBFERR	Calls ICHTRCH to process terminal object-time errors.
IBFINT	Processes program interruptions.
FBKSP	Implements the BACKSPACE source statement.

Table 37. 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.

Chart 26. IHCFIOSH Overall Logic

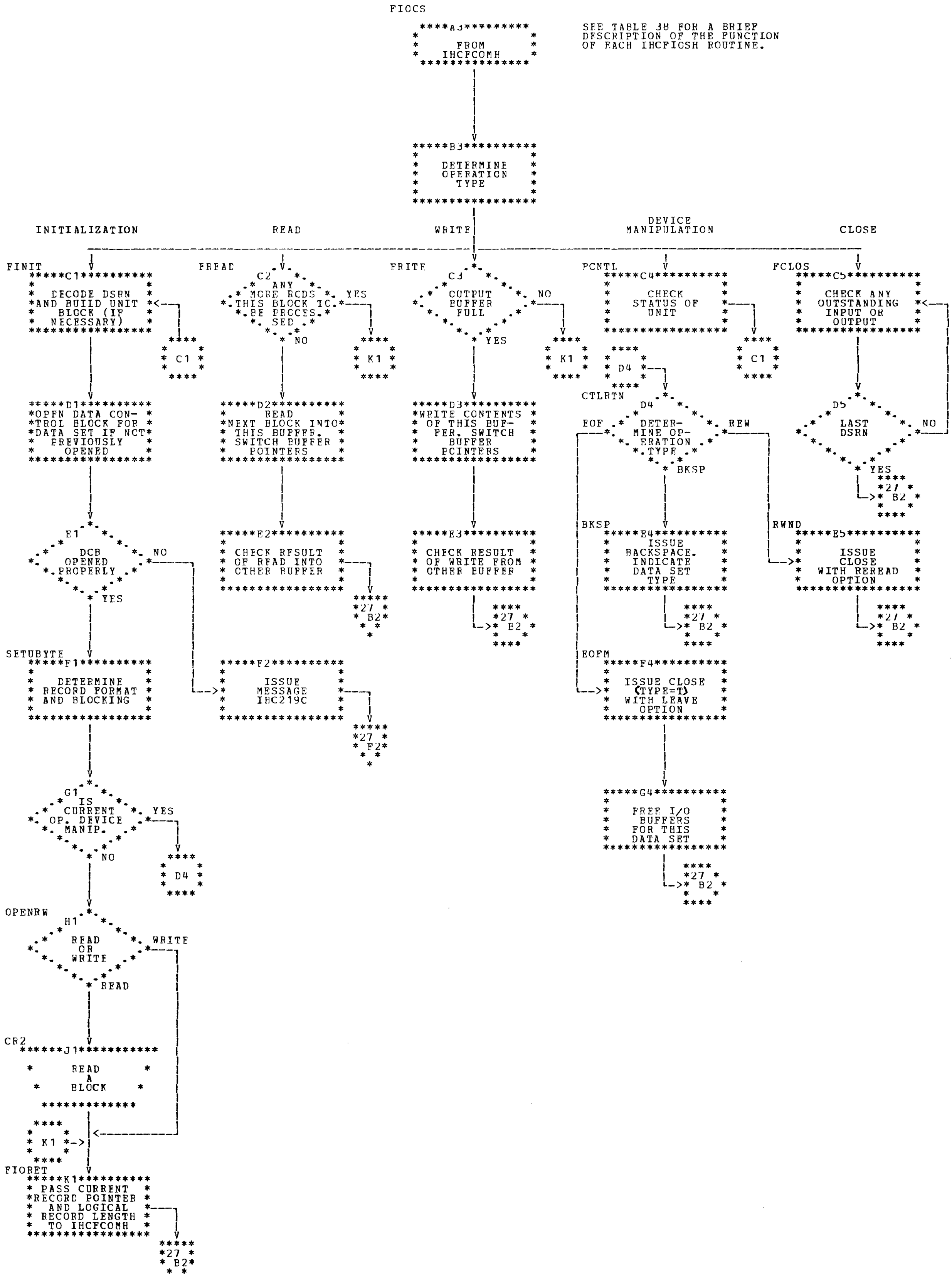


Chart 27. Execution-Time I/O Recovery Procedure

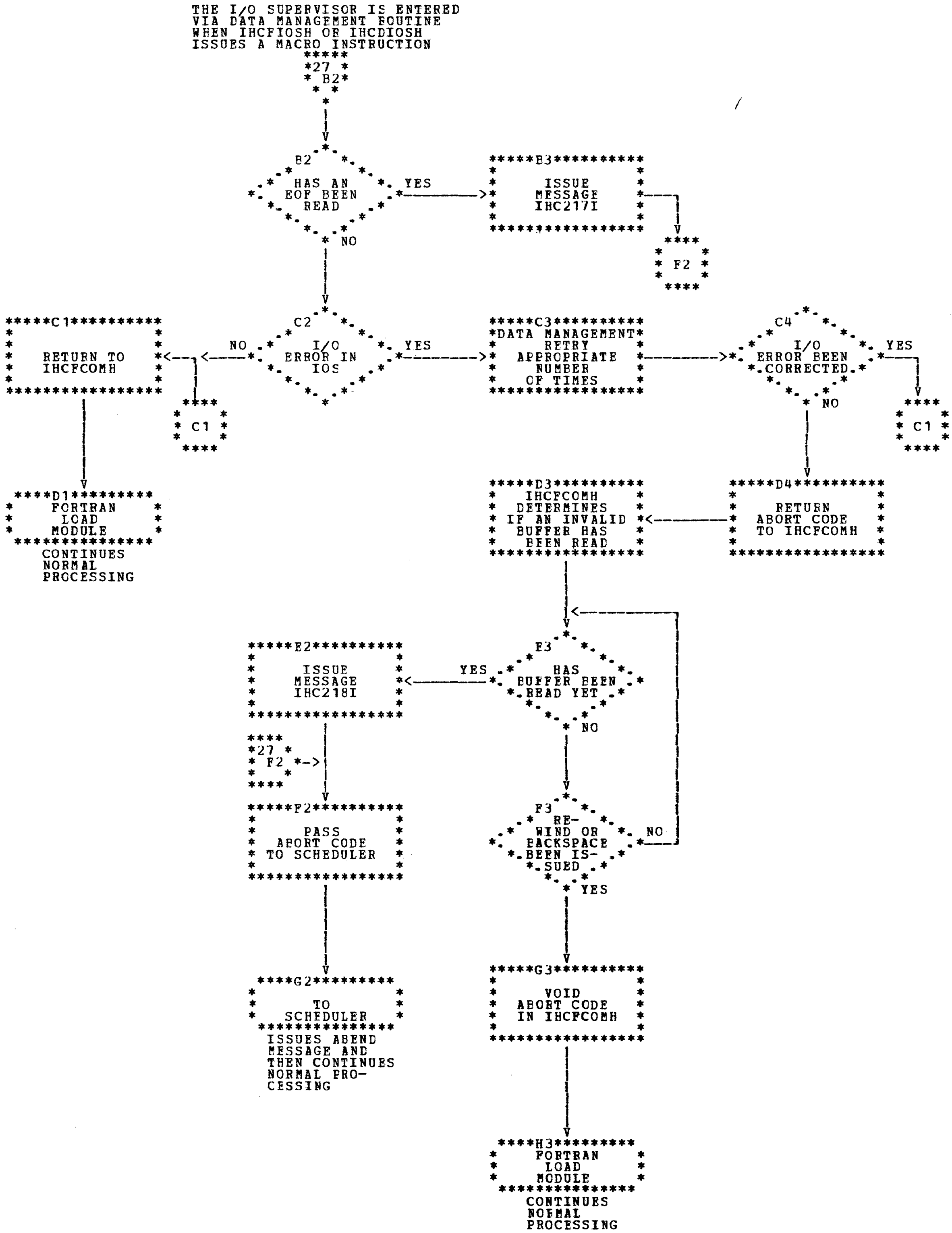


Chart 28. IHCDIOSH Overall Logic - File Definition Section

```
                            IHCIBERH IS
                            ENTERED VIA
                            CALLING SE-
                            QUENCES GEN-
                            ERATED AT
 ****A3*********            COMPILE-TIME
*    FORTRAN    *
*     LOAD      *
*    MODULE     *
 ***************
        |
        |
        |
        |
        v
*****B3**********
*               *
*OBTAIN INTERNAL*
*SEQUENCE NUMBER*
*     (ISN)     *
*               *
*****************
        |
        |
        |
        |
        v
*****C3**********
*               *
*  CONVERT ISN  *
*  TO DECIMAL   *
*    FORMAT     *
*               *
*****************
        |
        |
        |
        |
        v
*****D3**********
*   BRANCH TO   *
*  IHCFCOMH TO  *
*  HANDLE THE   *
*  WRITING OF   *
* ERROR MESSAGE *
*****************
        |
        |
        |
        |
        v
 ****E3*********
*  IBEXIT RTN   *
*      OF       *
*   IHCFCOMH    *
 ***************
```

Chart 29. IHCDIOSH Overall Logic - File Initialization, Read, Write, and Termination Sections

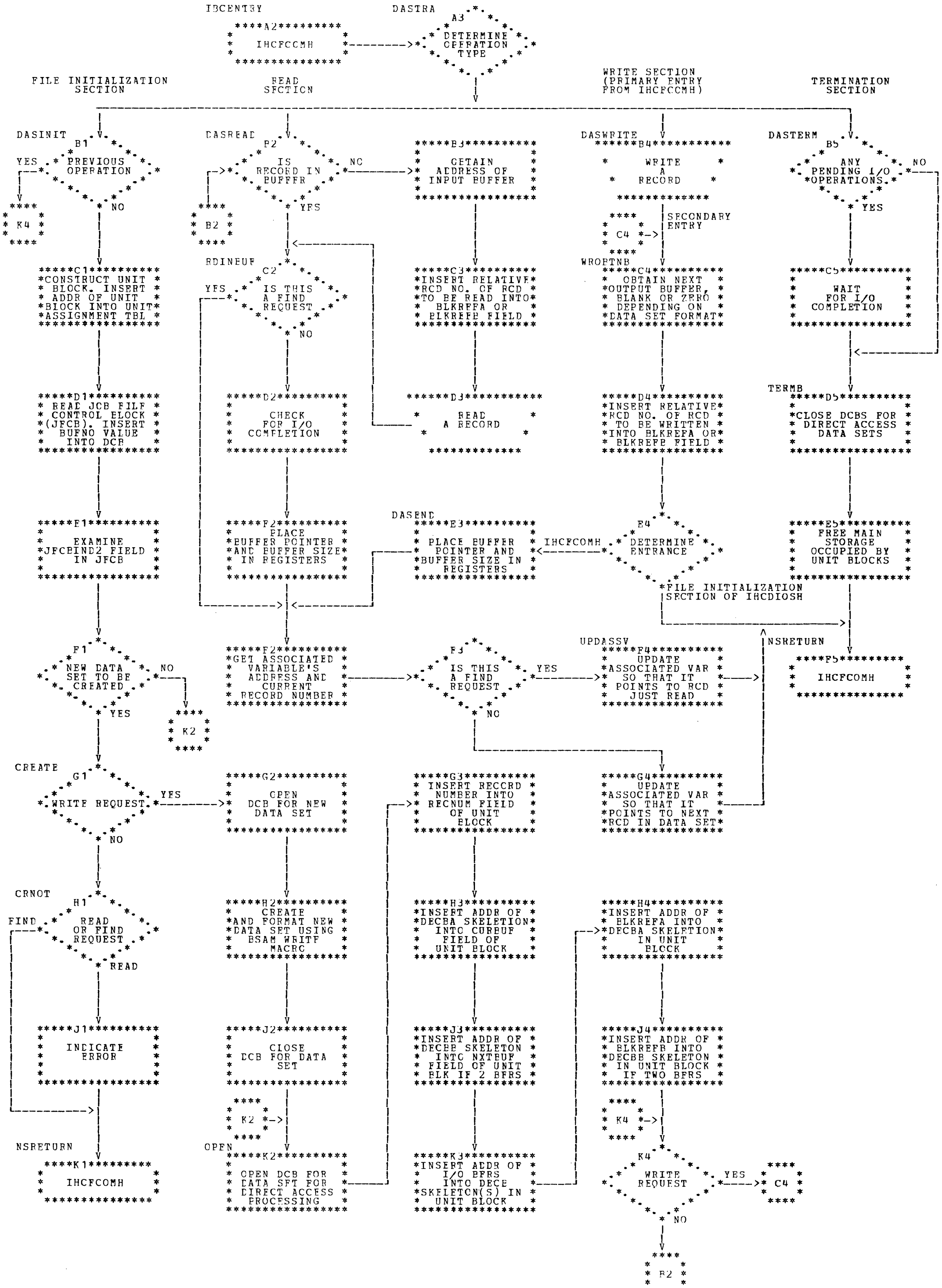


Table 38. IHCFIOSH Routine Directory

Routine	Function
FCLOS	CHECKs double-buffered output data sets.
FCNTL	Services device manipulation requests.
FINIT	Initializes unit and data set.
FREAD	Services read requests.
FRITE	Services write requests.

Table 39. IHCDIOSH Routine Directory

Routine	Function
DASDEF	Processes DEFINE FILE statements: enters address of parameter lists into unit assignment table, checks for redefinition of direct access unit numbers, and establishes addressability for IHCDIOSH within IHCFCOMH.
DASINIT	Constructs unit blocks for nonopened direct access data sets, creates and formats new direct access data sets, and opens data control blocks for direct access data sets.
DASREAD	Reads physical records, passes buffer pointers and buffer size to IHCFCOMH, and updates the associated variable.
DASTERM	Checks pending I/O operations, closes direct access data sets, and frees main storage occupied by unit blocks.
DASTRA	Determines operation type and transfers control to appropriate routine.
DASWRITE	Writes physical records, provides IHCFCOMH with buffer space, and updates the associated variable.

Chart 30. IHCIBERH Overall Logic

```
NOTE--                                          DIOCS

THE FILE DEFINITION                              ****A3*********      SEE TABLE 39 FOR A
SECTION IS ENTERED                               * FORTRAN LOAD  *    BRIEF DESCRIPTION OF THE
FROM THE FORTRAN                                 *    MODULE     *    FUNCTION OF EACH IHCDIOSH
LOAD MODULE VIA A                                *               *    ROUTINE.
COMPILER-GENERATED                                ***************
CALLING SEQUENCE.                                        |
                                                         |
                                                         v
                                                *****B3**********
                                                *   GET FIRST   *
                                                *  UNIT NUMBER  *
                                                *  (DSRN) FROM  *
                                                *PARAMETER LIST *
                                                *               *
                                                *****************
                                                         |
                                                         |<--------------------------.
                                                         v                           |
                                                *****C3**********                    |
                                                *  INSERT UNIT  *                    |
                                                *   NUMBER'S    *                    |
                                                *PARAMETER LIST *                    |
                                                *ADDRESS IN UNIT*                    |
                                                *ASSIGNMENT TBL *                    |
                                                *****************                    |
                                                         |                           |
                                                         v                           |
                                                        .*.                          |
                                                   D3 *.   *.               *****D4**********
                                                   .*         *.            *   GET NEXT    *
                                                .* LAST UNIT  *. NO         *  UNIT NUMBER  *
                                               *.  NUMBER IN  .*---------->*  (DSRN) FROM  *
                                                 *.PARAMETER.*              *PARAMETER LIST *
                                                   *.LIST .*                *               *
                                                     *. .*                  *****************
                                                       * YES
                                                       |
                                                       |
                                        DELAST         v
                                                *****E3**********
                                                *   ESTABLISH   *
                                                *LINKAGE BETWEEN*
                                                * IHCDIOSH AND  *
                                                *    IHCFCOMH   *
                                                *               *
                                                *****************
                                                         |
                                                         v
                                                 ****F3*********
                                                *    FORTRAN    *
                                                *     LOAD      *
                                                *    MODULE     *
                                                 ***************

                                                CONTINUE NORMAL
                                                   PROCESSING
```

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 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 and 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 in 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.

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.

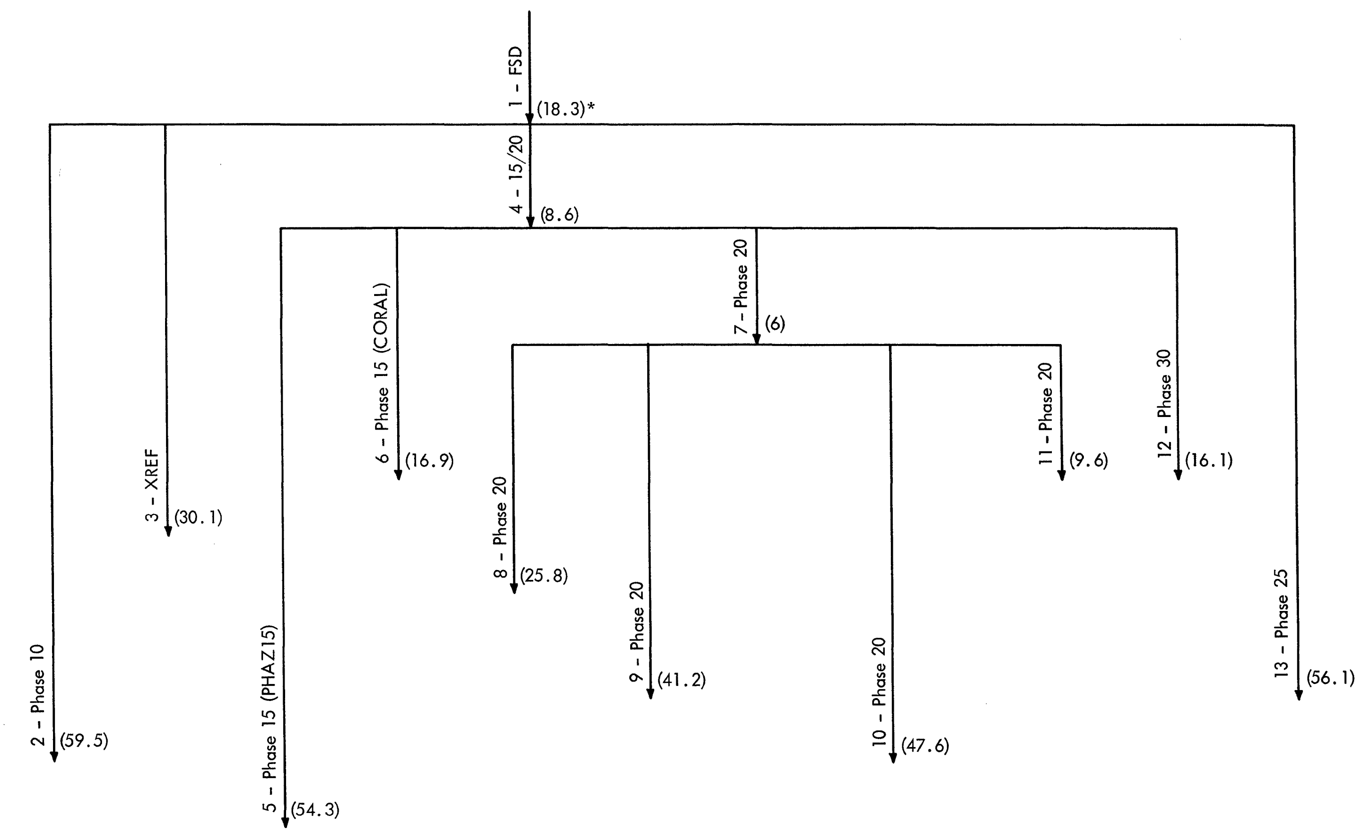
Each of the remaining 12 segments constitutes a phase or a major portion of a phase. Phase segments are overlaid as compiler processing requires the services of another segment.

Figure 62 illustrates the compiler's planned overlay structure. In the figure, each segment is identified by a number. Segments that originate from the same horizontal line overlay each other as needed. The figure also indicates the approximate size (in bytes) of each segment.

The longest path[1] of this structure is formed by segments 1, 4, and 5 because,

----

[1]A path consists of a segment, all segments between it and the root segment, and the root segment.



*The number in parentheses times 1,000 equals the approximate segment length.

• Figure 62. Compiler Overlay Structure

when they are in main storage, the compiler requires approximately 81,000 bytes. Thus, the minimum main storage requirement for the compiler is approximately 82,000 bytes.

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 40. The remainder of this appendix is devoted to a discussion of the segments of the

• Table 40. 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, 50, 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 41.

• Table 41. Segment - 1 Composition

Control Section	Entry Point(s)
IEKATB	
IEKAA01	PAGEHEAD
ADCON-IEKAAD	
PUTOUT-IEKAPT	PUTOUT
IEKATM	PHAZSS,PHASB,TST,PHASS, TSP,TOUT
DCLIST-IEKTDC	IEKTDC
AFIXPI-IEKAFP	FIXPI
SYSTAB-IEKTAB	IEKTAB
IEKAA00	IEKAGC,ENDFILE,IEKAA9
IEKFIOCS	FIOCS#,FIOCS
IEKFCOMH	IBCOM#,IBCOM
IEKTLOAD	IEKUSD,ESD,TXT,IEKTXT, RLD,IEKURL,IEND,IEKUND
ERCOM-IEKAER	
IEKAAA	

Segment 2: This segment is phase 10. The origin of the segment is immediately after 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 42.

• Table 42. Segment - 2 Composition

Control Section	Entry Point(s)
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
IEKKCS	IEKKOS
XICOP-IEKCIO	IEKCIO
PUTX-IEKCPX	IEKCPX
XDATA-IEKCDT	IEKCDT
GETWD-IEKCGW	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 after segment 1. If the XREF option is chosen, segment 3 overlays segment 2. If the XREF option is not selected, segment 3 is not used and segment 2 is overlaid by segment 4.

Segment 4: This segment is considered a portion of both phases 15 and 20. It contains data areas used by both phases. Included in this segment are RMAJOR-IEKJA4, CMAJOR-IEKJA2, the full register assignment tables, and phase 15/20 work areas. The origin of segment 4 is immediately after segment 1. Segment 4 is overlaid by segment 13 if abortive errors are not encountered during the processing of phases 10 and 15. The composition of segment 4 is illustrated in Table 43.

• Table 43. 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 after segment 4. Segment 5 is overlaid by segment 6. The composition of segment 5 is illustrated in Table 44.

•Table 44. Segment - 5 Composition

Control Section	Entry Point(s)
IEKLTB	
LOOKER-IEKLOK	
GENRTN-IEKJGR	IEKJGR
FUNRDY-IEKJFU	IEKJFU
CONSTV-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	

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 after segment 4. Segment 6 overlays segment 5 and is overlaid by segment 7 if syntactical errors are not encountered by phases 10 and 15. If errors are present, segment 6 is overlaid by segment 12. The composition of segment 6 is illustrated in Table 45.

•Table 45. Segment - 6 Composition

Control Section	Entry Point(s)
DFILE-IEKTDF	IEKTDF
NLIST-IEKTNL	IEKTNL
CORAL-IEKGCR	IEKGCR
NDATA-IEKGDA	IEKGDA
EQVAR-IEKGEV	IEKGEV
IEKGCZ	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 after segment 4. Segment 7 overlays segment 6 if source module errors are not encountered by phases 10 and 15. If errors are encountered, segment 7 overlays segment 12 after its processing is completed, only if errors encountered are not serious enough to cause deletion of the compilation. The composition of segment 7 is illustrated in Table 46.

•Table 46. Segment - 7 Composition

Control Section	Entry Point(s)
LPSEL-IEKPLS	IEKPLS
IEKARW	
TARGET-IEKPT	IEKPT
GETDIK-IEKPGK	IEKPGK,IEKPGC,IEKPIV, IEKPFT,TOFL
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 after segment 7. The composition of segment 8 is illustrated in Table 47.

•Table 47. Segment - 8 Composition

Control Section	Entry Point(s)
SRPRIZ-IEKQAA	IEKQAA,IEKQAB
TOPO-IEKPO	IEKPO
IEKPTB	IEKPTB
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 after 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 48.

•Table 48. Segment - 9 Composition

Control Section	Entry Point(s)
KORAN-IEKQKO	IEKQLO
WRITEX-IEKQWT	IEKQWT
CIRCLE-IEKQCL	IEKQCL,IEKQF
PERFOR-IEKQPF	IEKQPF
TYPLOC-IEKQTL	IEKQTL,IEKQIT
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 after segment 7. The composition of segment 10 is illustrated in Table 49.

•Table 49. 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 after 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 50.

•Table 50. 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 after segment 4. Segment 12 overlays segment 6 if syntactical errors are encountered during the processing of phases 10 and 15. If the errors detected by these phases are not serious enough to cause deletion of the compilation, segment 12, after its processing is completed, is overlaid by segment 7. The composition of segment 12 is illustrated in Table 51.

•Table 51. 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 after segment 1. Segment 13 overlays segment 4. The composition of segment 13 is illustrated in Table 52.

•Table 52. Segment - 13 Composition

Control Section	Entry Point(s)
MANGN2-IEKVM2	IEKVM2
PACKER-IEKTPK	IEKTPK
LABEL-IEKTLB	IEKTLB
RETURN-IEKTRN	IEKTRN
FNCALL-IEKVFN	IEKVFN
GOTCKK-IEKWKK	IEKWKK
LISTER-IEKTLS	IEKTLS
STOPPR-IEKTSR	IEKTSR
ENTRY-IEKTEN	IEKTEN
CGNDTA-IEKWCN	
BRLGL-IEKVBL	IEKVBL
IOSUB-IEKTIS	IEKTIS
PROLOG-IEKTPR	IEKTPR
MAINGN-IEKTA	IEKTA
TNTXT-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 (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 a value, m, for the PHASE SWITCH (refer to Appendix C of the above referenced publication). This value is in hexadecimal and indicates which phase of the compiler was in control when the error occurred. The value for m may be any one of the following:

m	Phase
1	Phase 10
4	Phase 15 (PHAZ15)
8	Phase 15 (CORAL)
10	Phase 20
20	Phase 25
40	Phase 30

Message number	Routine in which message number is generated	Phase in which message number is generated
IEK002I	XCLASS-IEKDCL	PHASE 10
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	
IEK012I	CSORN-IEKCCR	
IEK013I	XARITH-IEKCAR, PUTX-IEKCPX, CSORN-IEKCCR, XCLASS-IEKDCL	
IEK014I	XDATYP-IEKCDT, XSPECS-IEKCSP	
IEK016I	XGO-IEKCGO	
IEK017I	XGO-IEKCGO	
IEK019I	XGO-IEKCGO	
IEK020I	XGO-IEKCGO	
IEK021I	XGO-IEKCGO	

Message number	Routine in which message number is generated	Phase in which message number is generated
IEK022I	XGO-IEKCGO	PHASE 10
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	
IEK040I	XCLASS-IEKDCL	
IEK047I	XARITH-IEKCAR, XDATYP-IEKCDT	
IEK052I	DSPTCH-IEKCDP	
IEK053I	XARITH-IEKCAR, DSPTCH-IEKCDP	
IEK056I	XSUBPG-IEKCSR	
IEK057I	XSUBPG-IEKCSR	
IEK058I	XSUBPG-IEKCSR	
IEK059I	XSUBPG-IEKCSR	

IEK060I	XARITH-IEKCAR, DSPTCH-IEKCDP	PHASE 10
IEK062I	XSPECS-IEKCSP	
IEK064I	XTNDED-IEKCTN	
IEK065I	XTNDED-IEKCTN	
IEK066I	XTNDED-IEKCTN	
IEK067I	XTNDED-IEKCTN	
IEK069I	XSPECS-IEKCSP	
IEK070I	XSPECS-IEKCSP	
IEK072I	XSPECS-IEKCSP	
IEK073I	XSPECS-IEKCSP	
IEK074I	XSPECS-IEKCSP	
IEK077I	XTNDED-IEKCTN	
IEK078I	XTNDED-IEKCTN	
IEK079I	XTNDED-IEKCTN	
IEK080I	XTNDED-IEKCTN	
IEK081I	XTNDED-IEKCTN	
IEK082I	XTNDED-IEKCTN	
IEK083I	XTNDED-IEKCTN	
IEK084I	XTNDED-IEKCTN	
IEK093I	XDATYP-IEKCTN	
IEK094I	XDATYP-IEKCTN	
IEK095I	XDATYP-IEKCTN	
IEK096I	XDATYP-IEKCTN	
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	

IEK113I	XIOPST-IEKDIO	PHASE 10
IEK115I	XIOPST-IEKDIO	
IEK116I	XDO-IEKCDO	
IEK117I	DSPTCH-IEKCDP	
IEK120I	DSPTCH-IEKCDP	
IEK121I	XDATYP-IEKCDT	
IEK122I	XDATYP-IEKCDT	
IEK123I	XDATYP-IEKCDT	
IEK124I	XDATYP-IEKCDP	
IEK125I	XDATYP-IEKCDP	
IEK129I	XDATYP-IEKCDT	
IEK130I	XDATYP-IEKCDT	
IEK132I	XDATYP-IEKCDT	
IEK133I	XDO-IEKCDO	
IEK134I	XDO-IEKCDO	
IEK135I	XDO-IEKCDO	
IEK136I	XDO-IEKCDO	
IEK137I	XDO-IEKCDO	
IEK138I	XDO-IEKCDO	
IEK139I	DSPTCH-IEKCDP, XSPECS-IEKCSP, XDATYP-IEKCDT	
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	XSPECS-IEKCSP	
IEK150I	XSPECS-IEKCSP	
IEK151I	XSPECS-IEKCSP	
IEK152I	XSUBPG-IEKCSR	

IEK153I	XARITH-IEKCAR	PHASE 10
IEK156I	XIOOP-IEKCIO	
IEK157I	XARITH-IEKCAR	
IEK158I	XDO-IEKCDO	
IEK159I	XIOPST-IEKDIO	
IEK160I	XIOOP-IEKCIO, XDO-IEKCDO	
IEK161I	XIOOP-IEKCIO	
IEK163I	XDO-IEKCDO	
IEK165I	XIOOP-IEKCIO	
IEK166I	XIOOP-IEKCIO	
IEK167I	XARITH-IEKCAR, XSPECS-IEKCSP, XIOPST-IEKDIO, DSPTCH-IEKCDP, XSUBPG-IEKCSR	
IEK168I	XSUBPG-IEKCSR	
IEK169I	XICOP-IEKCIO	
IEK170I	XICOP-IEKCIO	
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	
IEK204I	XIOPST-IEKDIO	
IEK205I	XGO-IEKCGO	
IEK206I	XARITH-IEKCAR	
IEK207I	DSPTCH-IEKCDP	
IEK208I	DSPTCH-IEKCDP	
IEK211I	CSORN-IEKCCR	
IEK224I	XCLASS-IEKDCL, DSPTCH-IEKCDP	

IEK225I	DSPTCH-IEKCDP	
IEK226I	CSORN-IEKCCR	
IEK229I	XARITH-IEKCAR	
IEK302I	STALL-IEKGST	PHASE 10 (STALL-IEKGST) and PHASE 15 (CORAL)
IEK304I	STALL-IEKGST	
IEK305I	STALL-IEKGST	
IEK306I	STALL-IEKGST	
IEK307I	CORAL-IEKGCR	
IEK308I	STALL-IEKGST	
IEK310I	STALL-IEKGST	
IEK312I	STALL-IEKGST	
IEK314I	STALL-IEKGST	
IEK315I	STALL-IEKGST	
IEK318I	NDATA-IEKGDA	
IEK319I	NDATA-IEKGDA	
IEK322I	STALL-IEKGST	
IEK323I	STALL-IEKGST	
IEK332I	STALL-IEKGST	
IEK334I	STALL-IEKGST	
IEK350I	NDATA-IEKGDA	
IEK352I	NDATA-IEKGDA	
IEK353I	IEKGCZ	
IEK356I	STALL-IEKGST	
IEK500I	STALL-IEKGST	
IEK501I	UNARY-IEKKUN (EXPON)	
IEK502I	UNARY-IEKKUN (EXPON)	
IEK503I	BLTNFN-IEKJBF	
IEK505I	PHAZ15-IEKJA	
IEK506I	ALTRAN-IEKJAL	
IEK507I	BLTNFN-IEKJBF	
IEK508I	BLTNFN-IEKJBF	
IEK509I	PHAZ15-IEKJA	

IEK510I	ANDOR-IEKJAN	PHASE 15 (PHAZ15)
IEK511I	ANDOR-IEKJAN (IEKKNO)	
IEK512I	FINISH-IEKJFI	
IEK515I	RELOPS-IEKKRE	
IEK516I	FINISH-IEKJFI	
IEK520I	ALTRAN-IEKJAL	
IEK521I	ALTRAN-IEKJAL	
IEK522I	ALTRAN-IEKJAL	
IEK523I	ALTRAN-IEKJAL	
IEK524I	ALTRAN-IEKJAL	
IEK525I	ALTRAN-IEKJAL	
IEK526I	RELOPS-IEKRRE	
IEK527I	ANDOR-IEKJAN	
IEK528I	BLTNFN-IEKJBF	
IEK529I	DFUNCT-IEKJDF (IEKKPR)	
IEK530I	SUBADD-IEKKSA	
IEK531I	ALTRAN-IEKJAL	
IEK541I	DFUNCT-IEKJDF	
IEK542I	ALTRAN-IEKJAL	
IEK550I	ALTRAN-IEKJAL DFUNCT-IEKJDF (IEKKPR)	
IEK55I	GENER-IEKLGN	
IEK560I	GENER-IEKLGN	

IEK573I	GENER-IEKLGN TXTLAB-IEKLAB, TXTREG-IEKLRG	PHASE 15 (PHAZ15)
IEK580I	ALTRAN-IEKJAL	
IEK581I	SUBMLT-IEKKSM	
IEK583I	TXTREG-IEKLRG	
IEK584I	MATE-IEKLMA	
IEK585I	FINISH-IEKJFI	
IEK600I	TOPO-IEKPO	PHASE 20
IEK610I	TOPO-IEKPO	
IEK631I	GETDIK-IEKPGK	
IEK650I	TOPO-IEKPO	
IEK660I	TOPO-IEKPO	
IEK670I	BAKT-IEKPB	
IEK671I	BIZX-IEKPZ	
IEK680I	RELCOR-IEKRRL	
IEK710I	STALL-IEKGST	PHASE 10 (STALL-IEKGST)
IEK720I	STALL-IEKGST	
IEK730I	STALL-IEKGST	
IEK740I	STALL-IEKGST	
IEK750I	STALL-IEKGST	
IEK760I	STALL-IEKGST	
IEK770I	STALL-IEKGST	
IEK780I	MAINGN-IEKTA	PHASE 25
IEK999I	IEKP30	PHASE 30
IEK001I	IEKP30	

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 appearing in Table 53, 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 53 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 12 (the sum of basic keyword values 4 and 8) is specified, the compiler will generate both PHAZ15 diagnostic information and CORAL diagnostic information.

Table 53. Basic TRACE Keyword Values and Output Produced

Basic Keyword Values	Output Produced
1	Phase 10 diagnostic information
2	Printout of the information table as it appears after the execution of STALL in Phase 10
4	PHAZ15 diagnostic information
8	CORAL diagnostic information
16	Phase 20 diagnostic information
32	Phase 25 diagnostic information
64	Printout of: 1. Intermediate text and information table as they appear after the execution of Phase 10. 2. Information table as it appears after the execution of STALL in Phase 15. 3. Intermediate text and information table as they appear after the execution of PHAZ15 in Phase 15. 4. Information table as it appears after the execution of CORAL in Phase 15. 5. 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 will be attempted.

APPENDIX J: FACILITIES USED BY THE COMPILER

The following statements, built-in functions, and facilities are used by the compiler to compile itself.

Facility	Purpose
STRUCTURE Statement	Provides a means of referring to fields within data structures which are located arbitrarily in main storage. The data structures may consist of sets of fields of mixed type and length.
LAND (a,b) built-in function	ANDs a and b to obtain a 4-byte logical result.
LOR (a,b) built-in function	ORs a and b to obtain a 4-byte logical result.
LXOR (a,b) built-in function	Exclusive ORs a and b to obtain a 4-byte logical result.
LCOMPL (a) built-in function	Takes the compliment of a to obtain a 4-byte logical result.
SHFTL (a,n) built-in function	Shifts a left n bit positions to obtain a 4-byte logical result.
SHFTR (a,n) built-in function	Shifts a right n bit positions to obtain a 4-byte logical result.
TBIT (c,k) built-in function	Tests bit k of value c to obtain a 4-byte logical result; on=. TRUE. , off=.FALSE.
MOD24 (d) built-in function	Sets the high-order byte of d to zero to obtain a 4-byte integer result.
BITON (v,k) bit-setting statement	Sets bit k of value v on.
BITOFF (v,k) bit-setting statement	Sets bit k of value v off.
BITFLP (v,k) bit-setting statement	Inverts bit k of value v.
The following error message may appear in connection with a STRUCTURE statement: IEK060I The expression has a structured variable without a subscript.	

The microfiche directory (Table 54) is designed to help you find named areas of code in the program listing, which is contained on microfiche cards at your installation. Microfiche cards are filed in alphameric order by object module name. If you wish to locate a control section, entry point, or table 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, a brief synopsis of its function (if it is a routine), its phase, its overlay segment and its flowchart ID (if applicable).

• Table 54. Microfiche Directory

Symbolic Name	Description	Object Module Name and CSECT Name	Phase	Overlay Segment	Chart ID (* - Only Mentioned in Chart)	Synopsis
ADMDGN-IEKVAD	Code generation routine	IEKVAD#	25	13	--	Table 14
AFIXPI	Entry point	IEKAFP	FSD	1	--	Table 6
ALTRAN-IEKJAL	Arithmetic translation routine	IEKJAL#	15	5	07	Table 9
ANDOR-IEKJAN	Text generation routine	IEKJAN#	15	5	07*	Table 9
BACMOV-IEKQBM	Text optimization routine	IEKQBM#	20	9	12	Table 12
BAKT-IEKPB	Structural determination routine	IEKQPB#	20	8	10*	Table 12
BITNFP-IEKVFP	Instruction generation routine	IEKVFP#	25	13	--	Table 14
BIZX-IEKPZ	MVX routine	IEKPZ#	20	8	10*	Table 12
BKDMP-IEKRBK	Printing routine	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
CGNDTA-IEKWCN	Array initialization routine	IEKWCN	25	13	--	Table 14
CIRCLE-IEKQCL	Utility subroutine	IEKQCL#	20	9	--	Table 13
CLASIF-IEKQCF	Utility subroutine	IEKQCF#	20	9	--	Table 13
CNSTCV-IEKKCN	Constant conversion routine	IEKKCN	15	5	--	Table 9

(Continued)

• Table 54. Microfiche Directory (Continued)

Symbolic Name	Description	Object Module Name and CSECT Name	Phase	Overlay Segment	Chart ID (* - Only Mentioned in Chart)	Synopsis
CORAL-IEKGCR	Control routine	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
DELTEX-IEKQDT	Entry point	IEKQMT#	20	9	--	
DFUNCT-IEKJDF	In-line and library function routine	IEKJDF#	15	5	--	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	--	
END-IEKUEN	Object module processing 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	--	Table 14
EQVAR-IEKGEV	Common and equivalence routine	IEKGEV#	15	6	09*	Table 9
ESD	Entry point	IEKTLCAD	FSD	1	--	
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
FINISH-IEKJFI	Statement processing routine	IEKJFI#	15	5	07*	Table 9
FIOCS, FIOCS#	Entry points	IEKFIOCS	FSD	1	--	
FIXPI, FIXPI#	Entry points	IEKAFP	FSD	1	--	
FNCALL-IEKVFN	Calling sequence generating routine	IEKVFN#	25	13		Table 14
FREE-IEKRFR	Local register assignment routine	IEKRFR#	20	10	--	Table 12

(Continued)

• Table 54. Microfiche Directory (Continued)

Symbolic Name	Description	Object Module Name and CSECT Name	Phase	Overlay Segment	Chart ID * - Cnly Mentioned in Chart	Synopsis
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
GENERTN-IEKJGR	Text entry routine	IEKJGR#	15	5	--	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
GOTCKK-IEKWKK	Branching routine	IEKWKK#	25	13	--	Table 14
IBCOM, IBCOM#	Entry points	IEKFCOMH	FSD	1	--	
IEKAAA	Communication table	IEKAAA	FSD	1	--	Table 6
IEKAAD	Internal adcon table	IEKAAD	FSD	1	--	Table 6
IEKAA00	Compiler initialization routine	IEKAA00	FSD	1	01*	Table 6
IEKAA01	Default options, &DDNAMES for compiler	IEKAA01	FSD	1	--	Table 6
IEKAA9	Compilation deletion routine	IEKAA9	FSD	1	--	Table 6
IEKAER	Error message table	IEKAER	FSD	1	--	Table 6
IEKAFP	Exponentiation routine	IEKAFP	FSD	1	--	Table 6
IEKAGC	Entry point	IEKAA00	FSD	1	02*	Table 6
IEKAPT	Service routine	IEKAPT	FSD	1	--	Table 6
IEKAREAD	Entry point	IEKCGC	10	2	--	Table 8
IEKATB	Diagnostic dump 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

(Continued)

• Table 54. Microfiche Directory (Continued)

Symbolic Name	Description	Object Module Name and CSECT Name	Phase	Overlay Segment	Chart ID (* - Only Mentioned in Chart)	Synopsis
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
IEKGCR	CORAL controlling routine	IEKCGR#	15	6	09	Table 9
IEKGCZ	Base and displacement routine	IEKGCZ#	15	6	09*	Table 9
IEKGMP	Storage map routine	IEKGMP#	25	13	20*	Table 14
IEKGST	Table entry and text generation utility routine	IEKGST#	10	2	04	Table 8
IEKIORTN	Entry point	IEKAA00	FSD	1	--	
IEKJA2	Backward connection table	IEKJA2	15/20	4	--	
IEKJA4	Forward connection table	IEKJA4	15/20	4	--	
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	--	
IEKPFT	Entry point	IEKPGK#	20	7	--	
IEKPGC	Entry point	IEKPGK#	20	7	--	
IEKP30	Controlling routine	IEKP30	30	12	22	Table 15
IEKP31	Error message writing routine	IEKP31#	30	12	22*	Table 15
IEKQAB	Entry point	IEKQAA#	20	8	--	
IEKQDT	Entry point	IEKQMT#	20	9	--	
IEKQF	Entry point	IEKQCL#	20	9	--	
IEKQMF	Entry point	IEKQCF#	20	9	--	
IEKQPX	Entry point	IEKQCF#	20	9	--	
IEKQYS	Entry point	IEKQXS#	20	9	--	

(Continued)

• Table 54. Microfiche Directory (Continued)

Symbolic Name	Description	Object Module Name and CSECT Name	Phase	Overlay Segment	Chart ID * - Only Mentioned in Chart	Synopsis
IEKQZS	Entry point	IEKQXS#	20	9	--	
IEKRAL	Entry point	IEKRFL#	20	10	--	
IEKRTF	Entry point	IEKRFL#	20	10	--	
IEKTDC	Listing routine	IEKTDC#	FSD	1	--	Table 6
IEKTDF	Define file statement routine	IEKTDF#	15	6	09*	Table 9
IEKTDT	Data statement routine	IEKTDT#	15	6	09*	Table 9
IEKTLOAD	ESD, TXT, RLD, and loader END record building routine	IEKTLOAD	FSD	1	--	Table 6
IEKTXT	Entry point	IEKTLOAD	FSD	1	--	
IEKUND	Entry point	IEKTLOAD	FSD	1	--	
IEKURL	Entry point	IEKTLOAD	FSD	1	--	
IEKUSD	Entry point	IEKTLOAD	FSD	1	--	
IEKXRF	XREF routine	IEKXRF	--	3	--	
IEKXRS	Utility routine for XREF	IEKXRS	10	2	--	Table 8
IEND	Entry point	IEKTLOAD	FSD	1	--	
INVERT-IEKPIV	Entry point	IEKPGK#	20	7	--	
IOSUB-IEKTIS	Calling sequence generating routine	IEKTIS#	25	13	20*	Table 14
IOSUB2-IEKTIO	Calling sequence generating routine	IEKTIO#	25	13	--	Table 14
KORAN-IEKGKO	Utility subroutine	IEKQKO	20	9	13*	Table 13
LABEL-IEKTLB	Statement number routine	IEKTLB#	25	13	20*	Table 14
LABTLU-IEKCLT	Statement number utility routine	IEKCLT#	12	2	--	Table 8
LISTER-IEKTLS	Listing routine	IEKTLS#	25	13	--	Table 14
LOC-IEKRL1	Register assignment data	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	09*	Table 13
LPSEL-IEKPLS	Control routine	IEKPLS#	20	7	10*	Table 12
MAINGN-IEKTA	Control routine	IEKTA#	25	13	20	Table 14

(Continued)

•Table 54. Microfiche Directory (Continued)

Symbolic Name	Description	Object Module Name and CSECT Name	Phase	Overlay Segment	Chart ID * - Only Mentioned in Chart	Synopsis
MAINGN2-IEKVM2	Control routine	IEKVM2#	25	13	--	Table 14
MATE-IEKLMA	MVS, MVF, MVX routine	IEKLMA#	15	5	--	Table 9
MODFIX-IEKQMF	Entry point	IEKQCF#	20	9	--	
MOVTEX-IEKQMT	Utility subroutine	IEKQMT#	20	9	--	Table 13
MSGWRT-IEKP31	Error message writing routine	IEKP31#	30	12	22*	Table 14
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	--	
PAREN-IEKKPA	Parenthesis routine	IEKKPA#	15	5	07*	Table 9
PARFIX-IEKQPX	Entry point	IEKQCF#	20	9	--	Table 13
PERFOR-IEKQPF	Constant routine	IEKCPF#	20	9	--	Table 13
PHASB	Entry point	IEKATM	FSD	1	--	
PHASS	Entry point	IEKATM	FSD	1	--	
PHAZSS	Entry point	IEKATM	FSD	1	--	
PH10-IEKCAA	Common data area	IEKCAA	10	2	--	Table 8
PLSGEN-IEKVPL	Code generation routine	IEKVPL#	25	13	--	Table 14
PROLOG-IEKTPR	Prologue generating routine	IEKTPR#	25	13	21*	Table 14
PUTOUT	Entry point	IEKAPT	FSD	1	--	
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	22*	Table 14
RLD	Entry point	IEKTLOAD	FSD	1	--	

(Continued)

•Table 54. Microfiche Directory (Continued)

Symbolic Name	Description	Object Module Name and CSECT Name	Phase	Overlay Segment	Chart ID (* - Only Mentioned in Chart)	Synopsis
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 12
SSTAT-IEKRSS	Status setting routine	IEKRSS#	20	11	10*	Table 12
STALL-IEKGST	Table entry and text generation utility routine	IEKGST#	10	2	04	Table 8
STOPPER-IEKTSR	STOP and PAUSE statement routine	IEKTSR#	25	13	--	Table 14
STTEST-IEKKST	Replacement statement routine	IEKKST#	15	5	07*	Table 9
STXTR-IEKRSX	Control 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
SUBMULT-IEKKSM	Subscript computation routine	IEKKSM#	15	5	07*	Table 9
SUBSUM-IEKQSM	Operand and operand value replacement routine	IEKQSM#	20	9	--	Table 12
TARGET-IEKPT	Loop and back target routine	IEKPT#	20	7	10*	Table 12
TENTXT-IEKVTN	Statement processing and label map routine	IEKVTN#	25	13	20*	Table 14
TIMERC	Entry point	IEKATM	FSD	1	--	
TNSFM-IEKRTF	Entry point	IEKRFL#	20	10	--	
TOPO-IEKPO	Back dominator routine	IEKPO#	20	8	10*	Table 12
TOUT	Entry point	IEKATM	FSD	1	--	
TSP	Entry point	IEKATM	FSD	1	--	
TST	Entry point	IEKATM	FSD	1	--	
TSTSET-IEKVTS	Code generation routine	IEKVTS#	25	13	--	Table 14
TXT	Entry point	IEKTLOAD	FSD	1	--	
TXTLAB-IEKLAB	Statement number processing routine	IEKLAB#	15	5	08*	Table 9
TXTREG-IEKLRG	Standard text processing routine	IEKLRG#	15	5	08*	Table 9

(Continued)

• Table 54. Microfiche Directory (Continued)

Symbolic Name	Description	Object Module Name and CSECT Name	Phase	Overlay Segment	Chart ID * - Only Mentioned in Chart	Synopsis
TYPLOC-IEKQTL	Strength reduction routine	IEKQTL#	20	9	13*	Table 12
UNARY-IEKKUN	Arithmetic triplet and exponentiation operator routine	IEKKUN#	15	5	07*	Table 9
UNRGEN-IEKVUN	Code generation routine	IEKVUN	25	13	--	Table 14
WIRTEX-IEKQWT	Diagnostic trace printing routine	IEKQWT#	20	9	--	Table 12
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	I/O statement routine	IEKCIO#	10	2	--	Table 8
XPELIM-IEKQXM	Common expression elimination routine	IEKQXM#	20	9	11	Table 12
XSCAN-IEKQXS	Local block scan routine	IEKQXS#	20	9	--	Table 12
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, and STRUCTURE table entry routine	IEKCTN#	10	2	--	Table 8
XIOPST-IEKDIO	ASSIGN, RETURN, FORMAT, PAUSE, BACKSPACE, REWIND, END FILE, STOP, and END table entry routine	IEKDIO#	10	2	--	Table 8

Y28-6642-1

IBM

International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, N.Y. 10601
[USA Only]

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
[International]