IBM

Y20-0012-0

SALES and SYSTEMS GUIDE

IBM System/360 Operating System FORTRAN IV (H) 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. Program Logic Manuals are intended for use by IBM customer engineers involved in program maintenance, and by system programmers involved in altering the program design. Program logic information is not necessary for program operation and use; therefore, distribution of this manual is limited to persons with program maintenance or modification responsibilities.

PREFACE

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 Operating System: Principles of Operation, Form A22-6821

IBM System/360 Operating System: FORTRAN IV, Form C28-6515-4

IBM System/360 Operating System: Introduction to Control Program Logic, Program Logic Manual, Form Z28-6605

IBM System/360 Operating System: FORTRAN IV 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 Z28-6604

IBM System/360 Operating System: Concepts and Facilities, Form C28-6535

IBM System/360 Operating System: Control Program Services, Form C28-6541

IBM System/360 Operating System: Linkage Editor, Program Logic Manual, Form Z28-6610

IBM System/360 Operating System: 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 described in sufficient detail to enable the reader to understand its operation, and to provide a frame of reference for the comments and coding supplied in the program listing. Common data, such as tables, blocks, and work areas are discussed only to the extent required to understand the logic of each component. Flowcharts and subroutine directories are included at the end of this section.

Following the second part are a number of appendixes, which contain reference material.

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

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.

Address comments concerning this publication to the IBM Corporation, Programming Systems Publications, Department D58, PO Box 390, Poughkeepsie, N.Y. 12602.

TABLES

CHARTS

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

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, or SYSPUNCH 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 0.

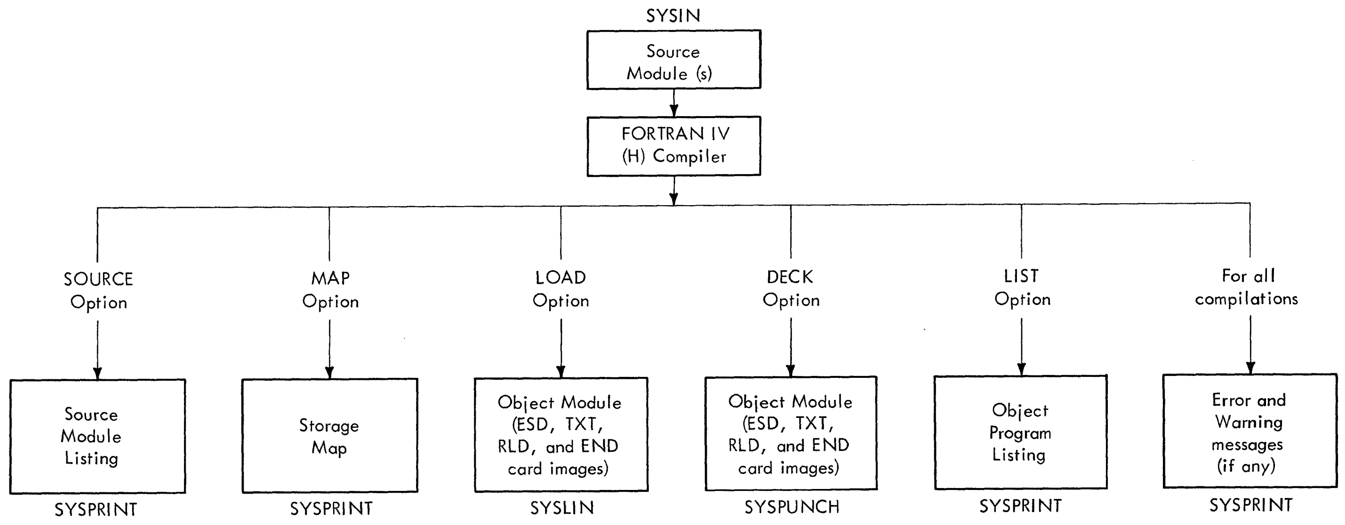


Figure 0. Input/Output Data Flow

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 (phase30).

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.

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.

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. 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. 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 the 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 three segments that perform the following functions:

- The first segment adds data to the information table about COMMON and EQUIVALENCE statements so that main storage space can be allocated correctly in the object module.
- The next segment tranlates 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 last 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 intermediate text (in operator-operand pair format) representing DATA statements to a variable-initial value form, which facilitates later assignment of a constant value to a variable. In addition, this segment produces a storage map if the MAP option is specified.

Phase 15 also passes to phase 30 the information needed to print the appropriate messages for the 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 degree of optimization desired.

If optimization has not been 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 a moderate amount 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 complete optimization has been 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 also records directly on the SYSPRINT data set messages describing any errors it detects.

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 text information.

Phase 25 places the object module resulting from the compilation on the SYS-

LIN data set if the LOAD option is specified, and on the SYSPUNCH data set if the DECK option is specified. Phase 25 also produces an object module listing on the SYSPRINT data set if the LIST 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.

STRUCTURE OF THE COMPILER

The FORTRAN IV (H) compiler is structured in a planned overlay fashion, which consists of 20 segments. The root segment is the FORTRAN system director. Each of the remaining 19 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 H.

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 performs compiler initialization, phase loading, storage distribution (including storage inventory), input/output request processing, compilation deletion, and compiler termination.

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 with a single entry is contained in a register. The entry in that list contains a pointer to a main storage area that contains an image of the options (e.g., SOURCE, MAP) specified for the compilation. The FSD scans this storage area and sets indicators to reflect the options specified. These indicators are placed into the communication table (refer to Appendix A, "Communication Table") during data field initialization.

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. It 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 GETCOR), 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. If main storage space is not available, the FSD deletes the compilation.

The main storage space available for building the information table or for collecting text entries is assembled into the FSD in the form of define storage (DS) statements. The distribution of the available storage by the FSD depends upon the phase requesting the storage. For this reason, the remainder of this discussion is divided into three parts: the first relating to phase 10, the second to phase 15, and the third to phase 20.

Phase 10 Storage

Phase 10 can use all of the available storage space for building the information table and for collecting text entries. At first, the FSD presents the entire block of available main storage space to phase 10 for use in building the information table. At each phase 10 request for main storage in which to collect text entries, the FSD reallocates a portion (i.e., a sub-block) of the storage (first allocated to the information table) for text collection, and returns to phase 10 either via the communication table or the storage area P10A (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, 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. 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 in which to collect text entries. (If the last information table entry and the sub-block to be allocated for text collection would overlap, the available storage is split, with one part being allocated for building the information table and the other for collecting text entries.)

The size of each sub-block allocated for the collection of phase 10 text entries depends upon the type of the text entries that are to be placed into the sub-block. All sub-blocks allocated to contain the same type of phase 10 text entries are of the same size.

Sub-blocks to contain phase 10 text 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 text entries may immediately follow a request for a sub-block to contain another type of text entries. Consequently, sub-blocks allocated to contain the same type of text 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. The manner in which the FSD keeps track of sub-blocks allocated to phase 10 is described in the following paragraph.

Phase 10 Storage Inventory: The FSD employs a pointer table and chains (see Figure 1) to keep track of the sub-blocks allocated for phase 10 text entries. If the sub-block allocated is the first to be used for the collection of a particular type of phase 10 text, the FSD places a pointer to that sub-block into the pointer table. After the initial link is established, the size of the sub-block is placed into the sub-block itself. If a second sub-block is allocated for the same purpose, the FSD places a pointer to it into the first word of the first sub-block allocated for that purpose. The size of the sub-block is then placed into the sub-block itself. If a third sub-block is allocated for the same purpose, the same procedure is followed, with a pointer to the third sub-block being placed into the first word of the second sub-block. Figure 1 illustrates this concept as applied to sub-blocks allocated to contain phase 10 normal, SF skeleton, and data text. (The pointer field of the last sub-block of each type is always zero.)

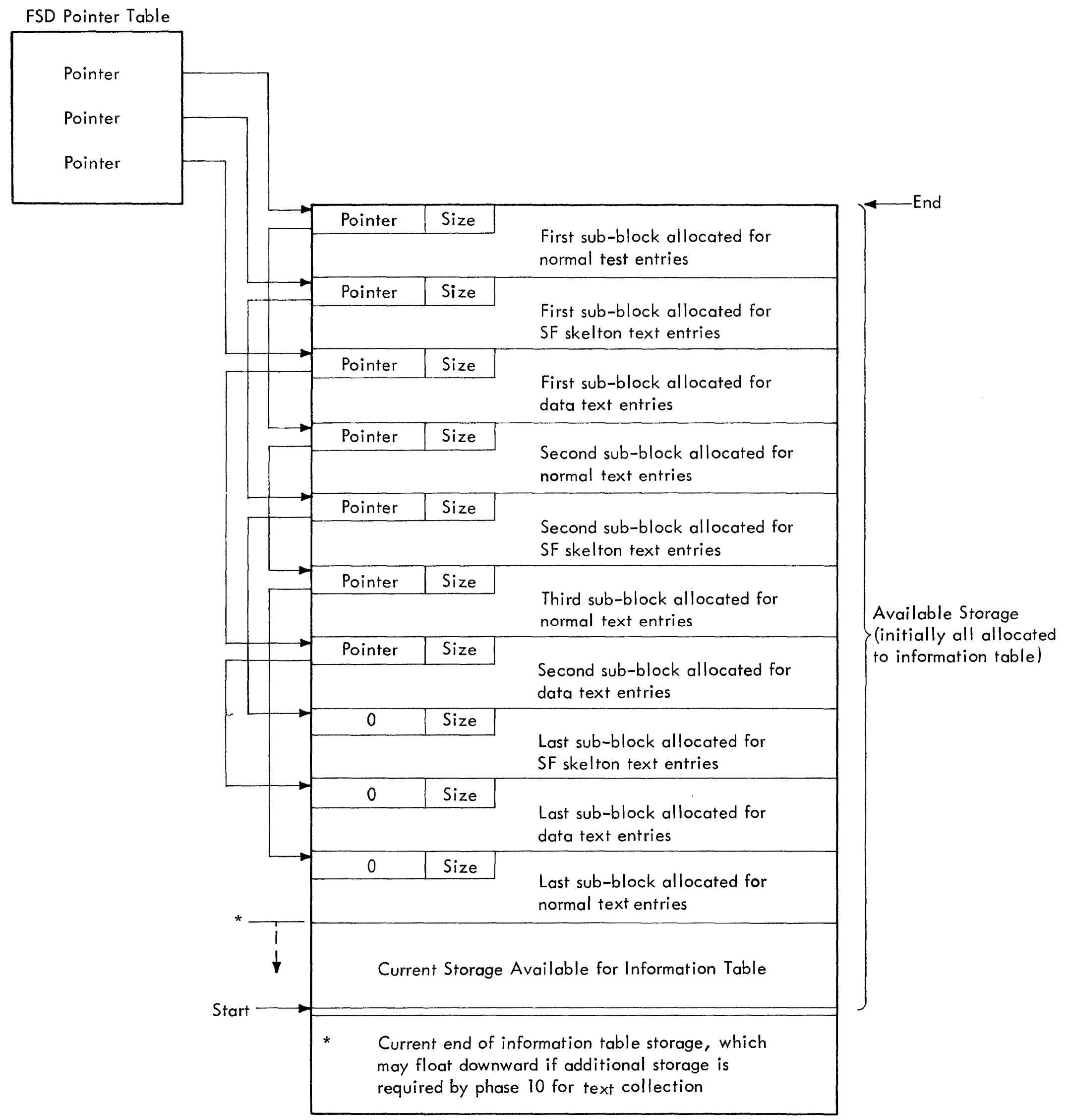

FSD Pointer Table
Pointer
Pointer
Pointer
Pointer
Size
First sub-block allocated for normal test entries
Pointer
Size
First sub-block allocated for SF skelton text entries
Pointer
Size
First sub-block allocated for data text entries
Pointer
Size
Second sub-block allocated for normal text entries
Pointer
Size
Second sub-block allocated for SF skelton text entries
Pointer
Size
Third sub-block allocated for normal text entries
Pointer
Size
Second sub-block allocated for data text entries
0
Size
Last sub-block allocated for SF skelton text entries
0
Size
Last sub-block allocated for data text entries
0
Size
Last sub-block allocated for normal text entries
*
Current Storage Available for Information Table
Start
End
Available Storage (initially all allocated to information table)
* Current end of information table storage, which may float downward if additional storage is required by phase 10 for text collection

Figure 1. Storage Inventory for Phase 10 Normal, SF Skeleton, and Data Text

Phase 15 Storage

Phase 15, in collecting the text entries that it creates, can use only those portions of main storage that are (1) unused by phase 10, and (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.

The main storage not used by phase 10 consists of:

- The portion between the last sub-block allocated to phase 10 for text collection and the end of the information table.
- Those portions of the sub-blocks allocated to phase 10 that do not contain text entries. (The last sub-block allocated to each type of phase 10 text may not be completely filled.)

After phase 10 processing is complete, the FSD splits the storage area between the last sub-block allocated to phase 10 and the last information table entry, allocates one part to the information table, and treats the other part as an unused text storage area. The individual portions of unused storage, excluding the portion allocated to the information table, are then chained together (see Figure 2). The first phase 15 request for storage for text collection is satisfied with the unused portion between the last sub-block allocated to phase 10 and the end of the information table. Pointers to both the beginning and end of the storage are passed to phase 15 via the communication table. Each subsequent phase 15 request for text area storage is satisfied with an unused portion of a phase 10 sub-block. (Sub-block portions are allocated in the order in which they are chained.) 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.

<u>Phase 15 Storage Inventory:</u> After the processing of PHAZ15, the second segment of phase 15, is completed, the FSD recovers the sub-blocks that were allocated to phase 10 normal and SF skeleton 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 third 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 data 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 data text is placed into that field. After the sub-blocks allocated for phase 10 data 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.

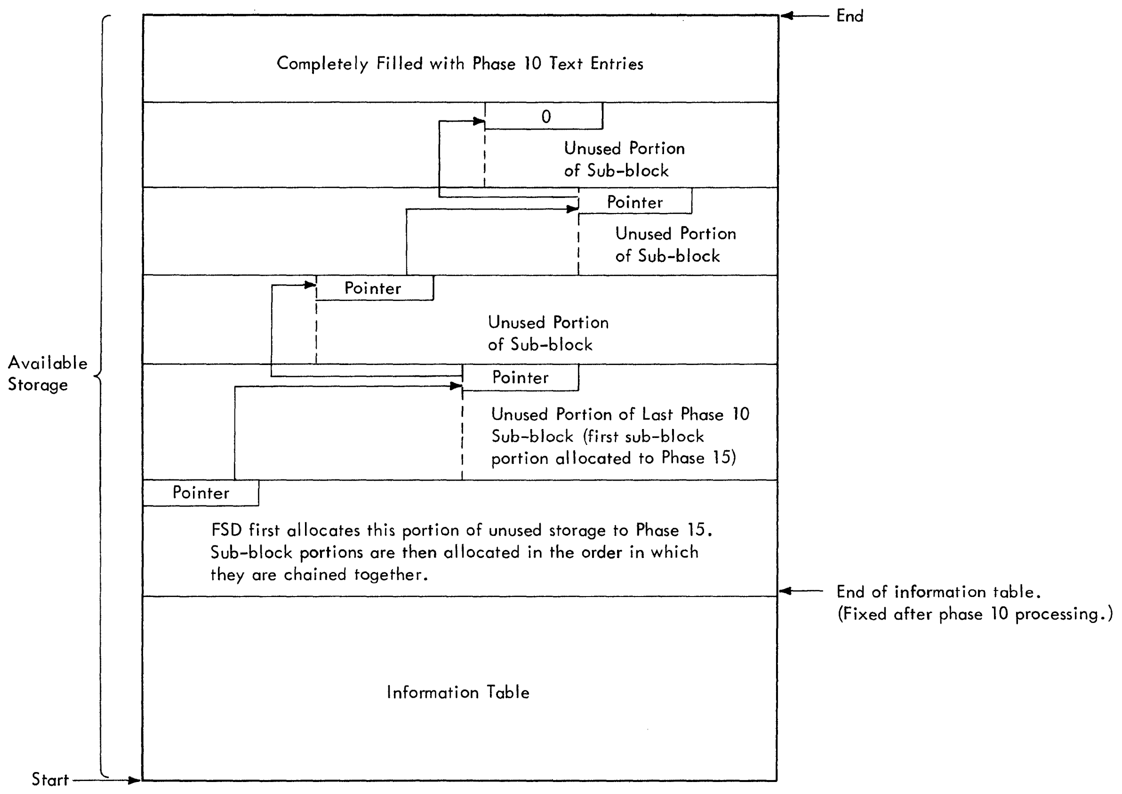

End
Completely Filled with Phase 10 Text Entries
0
Unused Portion of Sub-block
Pointer
Unused Portion of Sub-block
Pointer
Unused Portion of Sub-block
Pointer
Unused Portion of Last Phase 10 Sub-block (first sub-block portion allocated to Phase 15)
Pointer
FSD first allocates this portion of unused storage to Phase 15. Sub-block portions are then allocated in the order in which they are chained together.
End of information table. (Fixed after phase 10 processing.)
Information Table
Available Storage
Start

Figure 2. Chaining of Unused Text Area Main Storage

INPUT/OUTPUT REQUEST PROCESSING

The FSD routine IEKFCOMH receives the input/output requests of the compiler phases and submits them to BSAM (Basic 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$\underline{w}$ (output only), T$\underline{w}$, A$\underline{w}$, $\underline{w}$X, $\underline{w}$H, and Z$\underline{w}$ (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 IHCFCOMH/IHCFIOSH combination (see Appendixes E and F) to implement READ/WRITE statements requiring a format.

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 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. If the error was detected by phase 10, the FSD deletes the compilation by calling phase 10, which reads records (without processing them) until the END statement is encountered. Control is then returned to the FSD, which initializes the compiler for the next compilation. If the error was encountered in a phase other than phase 10, the FSD simply initializes the compiler for the next 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 initializes the compiler for the next compilation.

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 phase 10 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

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.

There are five types of intermediate text developed by phase 10. They are: normal, data, namelist, format, and statement function (SF) skeleton.

- Normal text is the intermediate text representation of source statements other than DATA, NAMELIST, 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.
- 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: dictionary, statement number/array table, common table, literal table, and branch table.

- 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). If a source module listing is requested, the record is recorded on an output data set (SYSPRINT). 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) 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 five groups of subroutines:

. Dispatcher subroutine.
. Preparatory subroutine.
. Keyword subroutines.
. Arithmetic subroutines.
. Utility subroutines.

Dispatcher Subroutine

The dispatcher subroutine (DSPTCH) controls phase 10 processing. Upon receiving control from the FSD, the DSPTCH subroutine initializes phase 10 processing and then calls the preparatory subroutine (GETCD) to read and prepare the first source statement. After the statement is prepared, control is returned to DSPTCH, which determines if a statement number is associated with the source statement being processed. If there is a statement number, the DSPTCH 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 subroutine then determines, from the 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 subroutine, which calls the preparatory subroutine to read and prepare the next source statement.

Preparatory Subroutine

The preparatory subroutine (GETCD) reads each source statement, 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 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 NCDIN in the format illustrated in Figure 3.

---

[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.

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)
Statement number	(5 words)
Packed source statement	(n words)
Group mark[2]	(1 word)
[1]For arithmetic statements and statement functions, this field points to the first character of the packed statement. [2]End of statement marker.	

Figure 3. Format of Prepared Source Statement

Keyword Subroutines

A keyword subroutine exists for each keyword source statement. A keyword source statement is any permissable 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.

Note: One keyword subroutine, namely that which processes the IMPLICIT statement, is not assigned to either of the above stated groups. The processing performed by this subroutine (XIMPC) is somewhat specialized. The function of this subroutine is defined in Table 8.

Table Entry Subroutines: Only four key word subroutines belong to this group (refer to Table 8). Each is associated with a COMMON, DIMENSION, EQUIVALENCE, or EXTERNAL key word statement.

The processing performed by these subroutines is similar. Each 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 these subroutines are:

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

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

Table entry and Text Subroutines: The keyword subroutines, other than those that are grouped as table entry subroutines, 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 subroutine, the keyword subroutine associated with the keyword statement being processed places a special operator into a text entry work 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 entry work area. The mode (e.g., integer, real) and type (e.g., negative constant, array) of the operand are then placed into the work area. The text entry thus developed is placed into the next available location in the sub-block allocated for text entries of the type being created.

Scanning is resumed and the next operator is obtained and placed into the text entry work 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 keywork 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 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 entry work 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 entry work 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.)

- Place the text entry into the next available location in the sub-block allocated for SF skeleton text.

- Resume scanning, obtain the next operator, and place it into the text entry work 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 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 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 and, if requested, verify that a converted constant is of integer mode.

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 processes common and equivalence entries in the common table, 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 three overlay segments, STALL, PHAZ15, and CORAL. Chart 04 shows the overall logic of the phase.

STALL processes both common and equivalence entries in the information table. It finds the maximum size of each common block, assigns locations to variables in each common block, and plans the storing of operands equated by EQUIVALENCE statements. It also determines the head of arrays referred to in EQUIVALENCE statements. (The head is the lowest-valued starting address of two or more arrays after their repositioning has been planned by equivalence processing.) CORAL later uses the head during the computation of relative addresses for variables and arrays.

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 the phase 20 optimization, includes tables on branching locations, and on constant and variable usage.

CORAL, the last overlay segment of phase 15, performs five functions. It first converts phase 10 data text to a form more easily evaluated by phase 25. 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 phase 25. CORAL also assigns address constants, when needed, to serve as address references for all operands. Lastly, as a user option, CORAL prints a storage map of named items (variables, arrays, and external references) as recorded in the information table.

STALL PROCESSING

STALL first rechains entries for variables in the dictionary by sorting alphabetically the entries within each chain. The rechaining frees storage in each entry for later use by CORAL.

As a second function, STALL checks the statement-number section of the information table, noting undefined statement numbers.

STALL then processes common entries in the information table. It computes the offset (displacement) of each variable in a common block from the start of the common block. The offsets are subsequently used to assign relative addresses to common variables. The offsets are recorded in the dictionary entries for the variables. The total size of each common block is also calculated. The block size is used by phase 25 to generate a control section for the common block.

Lastly, STALL processes equivalence entries in the information table. The processing plans the placing of the operands of each equivalence group at the same location in storage. During the processing STALL recognizes a variable that must be made equivalent to previously processed variables in common.

Chart 05 shows the overall processing of STALL.

Rechaining Entries for Variables

The STALL subroutine DCTSRT begins by rechaining entries for variables in the information table. Each dictionary entry created by phase 10 contains two chain address fields (refer to Appendix A, "Information Table Components"). DCTSRT frees one of the chain address fields for later use by CORAL. It does this by sorting alphabetically within each length grouping and then rechaining the entries. After the entries have been rechained, the dictionary consists of one chain for each variable-name length. The chains of entries describing symbols of 3 or less characters are arranged in descending alphabetic order, while the chains of entries describing symbols of 4 or more characters are arranged in ascending alphabetic order. As an integral part of rechaining, DCTSRT also constructs dictionary entries for the imaginary parts of complex variables and constants.

Checking for Undefined Statement Numbers

After subroutine DCTSRT has rechained the dictionary, subroutine LABSCN checks for undefined statement numbers. This action is taken to insure that every statement number that is referred to is also defined. LABSCN 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, LABSCN places an entry in the error table for later processing by phase 30.

Processing of Common Entries in the Information Table

After the statement numbers have been checked, subroutine COMN 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. COMN 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 (refer to Appendix A, "Common Table"). It 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 of Equivalence Entries in the Information Table

Subroutine EQU next 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"). EQU identifies and flags in their dictionary entries variables and arrays put into common via the EQUIVALENCE statement. It also error-checks the variables and arrays 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, subroutine EQU 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, EQU recognizes the combination of groups and modifies the dictionary entries for the variables to indicate the equivalence operations. EQU 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, subroutine TESTBN is given control to check that variables and arrays fall on boundaries appropriate to their defined types. If a variable or array is improperly aligned, TESTBN places an entry in the error table for processing by phase 30.

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 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 first 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.

---

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

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. PHAZ15 also gathers information on variables not first used and then defined. The foregoing usage information is recorded in the statement number text for each block for later use by phase 20.

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

After PHAZ15 has completed the previously mentioned processing, it reorders the statement number chain of the information table. The original order of statement numbers, as phase 10 recorded them, was in order of their occurrence in source statements as either definitions[2] 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 unit upon which the optimization and register assignment processes of phase 20 operate. A text block is a series of text entries that begin with the text entry for a statement number and end with the text entry that immediately precedes the text entry for the next statement number. 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 P2 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 P2 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 P2 field of the associated statement number text entry. All text blocks in the module are identified in this manner.

Figure 4 illustrates the concept of text blocking. In the figure, two text blocks

---

[1]If optimization is selected, phase 20 may further manipulate the phase 15 text.
[2]A statement number occurs as a definition when that statement number appears to the left of a source statement.

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.

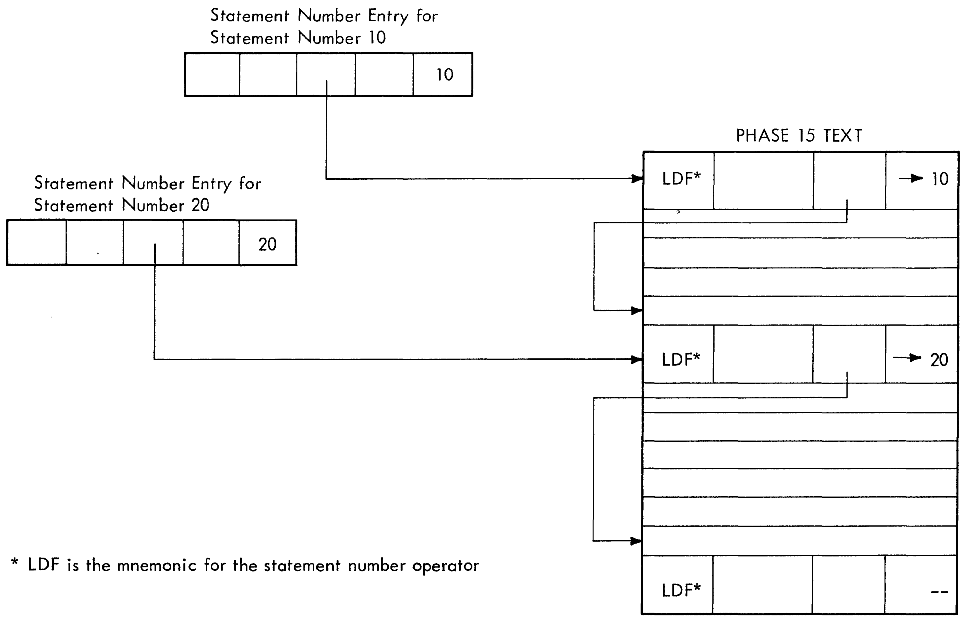

Statement Number Entry for Statement Number 10
10
PHASE 15 TEXT
LDF*
→ 10
Statement Number Entry for Statement Number 20
20
LDF*
→ 20
LDF*
--
* LDF is the mnemonic for the statement number operator

Figure 4. Text Blocking

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, ASSIGN, and GOTO statements and I/O datalist, as well as in arithmetic statements and statement functions.

When PHAZ15 detects a primary adjective code for a phase 10 text entry that may need arithmetic translation, it passes control to the arithmetic translator (ALTRAN). For simple expressions not having operator-operand pairs (terms) needing further special handling, the arithmetic translator reorders the expression so that the terms appear in phase 15 text in the order in which arithmetic operations should be performed.

While reordering expressions, the arithmetic translator determines whether or not a term needs special handling before it can be placed in the phase 15 text area. (Special handling is required for complex expressions, terms involving unary minuses (e.g., A=-B), subscripts, statement function references, etc.) If special handling is required, one or more subroutines are called to perform the needed processing.

After reordering and, if required, special handling, subroutine GENER places the processed text items in the phase 15 text area in four-part format.

REORDERING ARITHMETIC EXPRESSIONS: The reordering of arithmetic expressions is done by means of a pushdown table. This table is a last-in, first-out (LIFO) 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) 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 1.

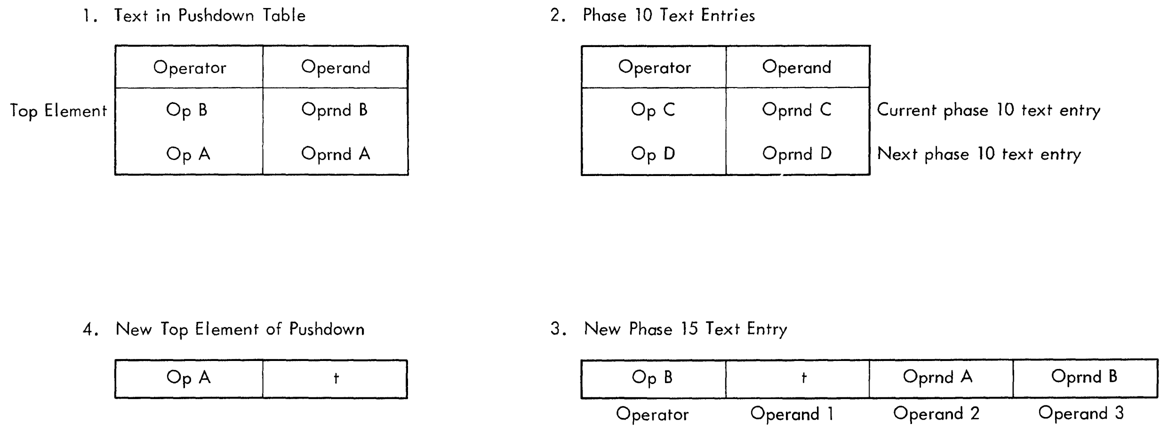
Table 1. Operators and Forcing Strengths

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

When the arithmetic translator (ALTRAN) 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 5, the number-1 section of the drawing shows the pushdown table at this time.

The operator of the next text entry (operator C--operand C at section 2) is compared with the top table element (operator B--operand B at section 1) in a similar manner.

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: 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 5. Text Reordering Via the Pushdown Table

Note that ALTRAN 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 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.

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

<u>SPECIAL PROCESSING OF ARITHMETIC EXPRESSIONS:</u> 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 library functions). 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 of two, commutative expressions, subscript expressions, routine or subprogram references, statement function references, and expressions involved in logical IF statements.

<u>Complex Expressions:</u> 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 subroutines UNARY and SWITCH, 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 of Two: Several kinds of special handling are provided by subroutines UNARY and EXPON for operations involving powers of two. Multiplication and division by powers of two are converted, respectively, to left and right 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 between -6 and +6 is converted to a series of multiplications (and a division into one, if necessary). 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 chain order in the dictionary.

Subscripts: Subroutines SBGLUT, SUBADD, SUBMLT, and SUBSCR perform subscript processing. Subscripted items are processed one at a time throughout the subscript. If the subscripted item 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, LIBRTN, NEGCHK, XPARAM, BLTNFN, and DFUNCT.

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 LIBRTN (for references to library routines) or subroutine BLTNFN (for references to in-line routines) then conducts a series of tests on the argument and perform the processing determined by the results of the tests.

If a function is not external and is in the IFUNTB table (refer to Appendix A, "Subprogram Table"), the IFUNT table is scanned to determine if the required routine is in-line. Then, the mode and number of arguments are tested. If the routine is in-line and the mode and number of arguments are as expected, DFUNCT 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. PHAZ15 provides in-line routines itself.[1] Instead of placing a special operator in text, PHAZ15 inserts a regular operator, such as the operator for AND or STORE.

If the mode and/or number of arguments in the 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, an error is assumed to

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

exist. Otherwise, the function is assumed to be external.

If a function is external (either used in an EXTERNAL statement or does not appear in the IFUNTB 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 subprogram or function call is then generated. The operator of the text entry is for an external function or subprogram reference. This 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 IFUNTB 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. (If none of the arguments are subscripted variables, the load address items are not required.) A text entry having a library function operator is generated. This entry points to the IFUNTB entry for the function. The text representation of the argument list is then generated and placed into the phase 15 text chain.

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

Logical Expressions: Subroutines ALTRAN, ANDOR, RELOPS, and NOT 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.

Gathering Constant/Variable Usage Information

During the conversion of the phase 10 text entries that follow the beginning of a text block (i.e., the text entries that follow a statement number definition) to phase 15 format, the PHAZ15 subroutine MATE gathers usage information for the variables and constants in that block. This information is required during the processing of the intermediate- and complete-optimized paths through phase 20 (refer to "Phase 20"). If optimized processing is not selected, this information is not compiled. Subroutine MATE 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 not first used and then defined (i.e., not busy-on-entry) within the text of the block.

Subroutine MATE 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 when it is first encountered in text.

Coordinates are assigned to variables and constants in the following manner:

- The first 59 unique variables and/or constants appearing in the text created by phase 15 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").

Subroutine MATE assigns the first variable or constant in phase 15 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. MATE assigns the second variable or constant a coordinate number of 3 and records its usage information in bit position 3 of the three fields. MATE continues this process until coordinate 60 has been assigned to a variable or constant. After coordinate number 60 has been assigned, MATE only assigns coordinates to the next 20 unique variables. (MATE 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. MATE does not assign coordinates to or gather usage information for unique variables encountered after coordinate number 80 has been assigned.

Subroutine MATE 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 MATE 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 MATE 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.

After a phase 15 text entry has been formed, subroutine MATE 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 (indicating that this is the first occurrence of the operand in the module), subroutine MATE assigns it the next coordinate, enters the coordinate number into the byte-C usage field of 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 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 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 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

---

[1]The coordinate 1 is assigned to simple variables that are made equivalent to variables of different modes, and to arrays.

is on, operand 1 of the text entry was previously encountered in the block as a use and therefore is not 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.

Gathering Forward Connection Information

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.

Subroutines TXTREG and TXTLAB record forward connection information 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 text entry (refer to "Text Blocking"), each RMAJOR entry points indirectly to a text block.

When PHAZ15 begins a new text block, it 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 thereby points to the first entry in RMAJOR in which the forward connection information for that block is to be recorded.

PHAZ15 then processes the phase 10 text entries following the statement number definition that caused PHAZ15 to begin the new text block. If it encounters a text entry for a IF, GO TO, or compiler generated branch following the statement number definition (and before the next), it passes control to subroutine TXTREG, which records in the next available entry in RMAJOR a pointer to the statement number entry for each statement number that may be branched to as a result of the execution of the IF, GO TO, or generated branch. A number of such text entries may be encountered in the text following the statement number definition and TXTREG records a pointer to the statement number entry for each statement number that may be branched to as a result of execution. (If two or more branches to the same statement number appear in the text following the statement number definition and before the next, TXTREG makes only entry in RMAJOR for the statement number to be branched to.)

When PHAZ15 encounters the next statement number definition, before beginning a new text block, it passes control to subroutine TXTLAB, which records in RMAJOR the fall-through connection information for the current block. This is a pointer to the statement number entry associated with the next statement number definition. The current text block may fall-through to the next and, hence, this connection information is required. The fall-through connection is flagged as the last for the current block. When the fall-through connection has been recorded, all the forward connection information for the text block has been gathered. Each entry that has been made in RMAJOR for the block, the first of which is pointed to by the statement number entry associated with the block and the last of which is flagged as such, points indirectly to a block to which that block may pass control.

Figure 6 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.

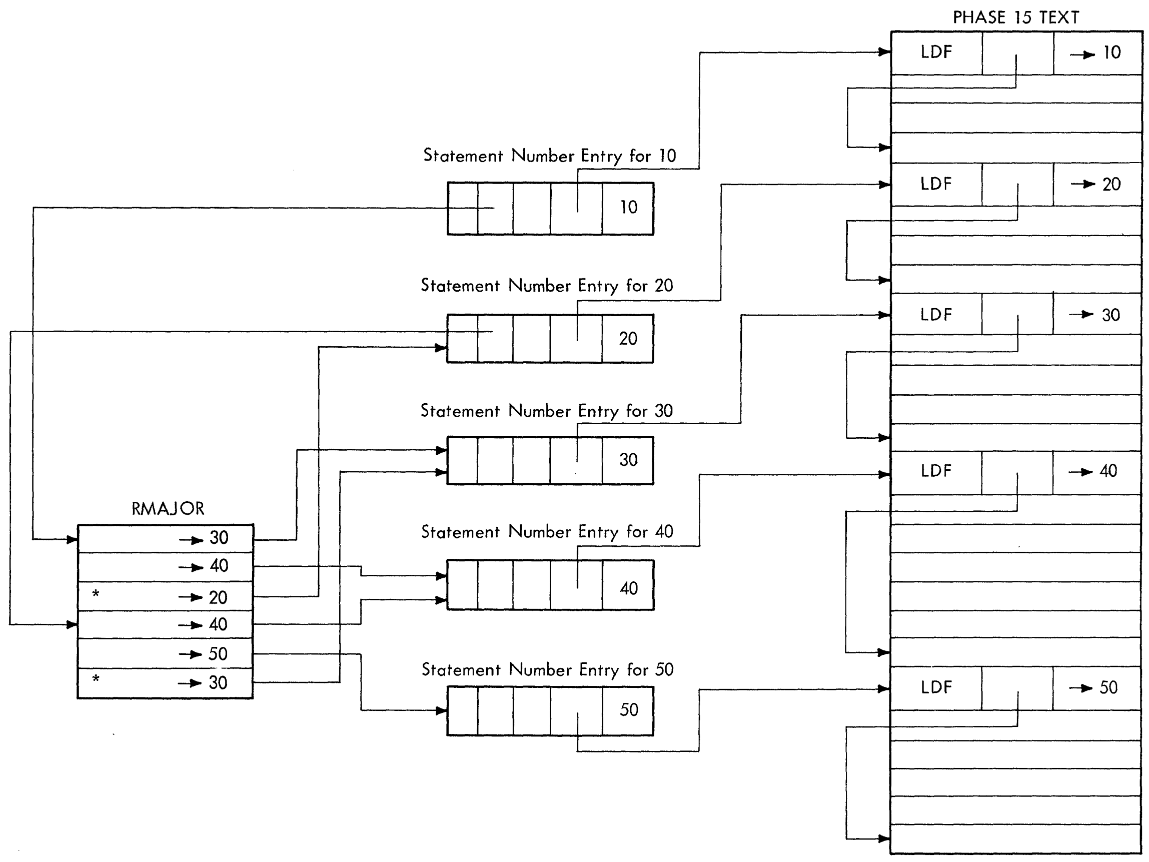

PHASE 15 TEXT
LDF
→ 10
Statement Number Entry for 10
10
LDF
→ 20
Statement Number Entry for 20
20
LDF
→ 30
Statement Number Entry for 30
30
LDF
→ 40
RMAJOR
→ 30
→ 40
* → 20
→ 40
→ 50
* → 30
Statement Number Entry for 40
40
Statement Number Entry for 50
50
LDF
→ 50

Figure 6. Forward Connection Information

Reordering the Statement Number Chain

After text blocking, arithmetic translation, and, if optimization has been specified, the gathering of constant/variable usage information have been completed, subroutine VSETUP 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, VSETUP uses this table to reorder the statement number entries. It places the first table pointer 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 VSETUP 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 VSETUP 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 VSETUP 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. Each CMAJOR entry made by VSETUP 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 VSETUP 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, VSETUP 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), VSETUP obtains, in turn, each entry in the statement number chain. (The entries are obtained in the order in which they are chained.) After VSETUP 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.) VSETUP 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 VSETUP records a pointer to its associated statement number entry in the next available entry in CMAJOR. VSETUP 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. VSETUP flags the last entry in CMAJOR for block I. When the statement number chain has been completely searched, VSETUP has gathered all the backward connection information for block I. Each entry that VSETUP 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 VSETUP 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 third segment of phase 15.

Figure 7 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.

- 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.

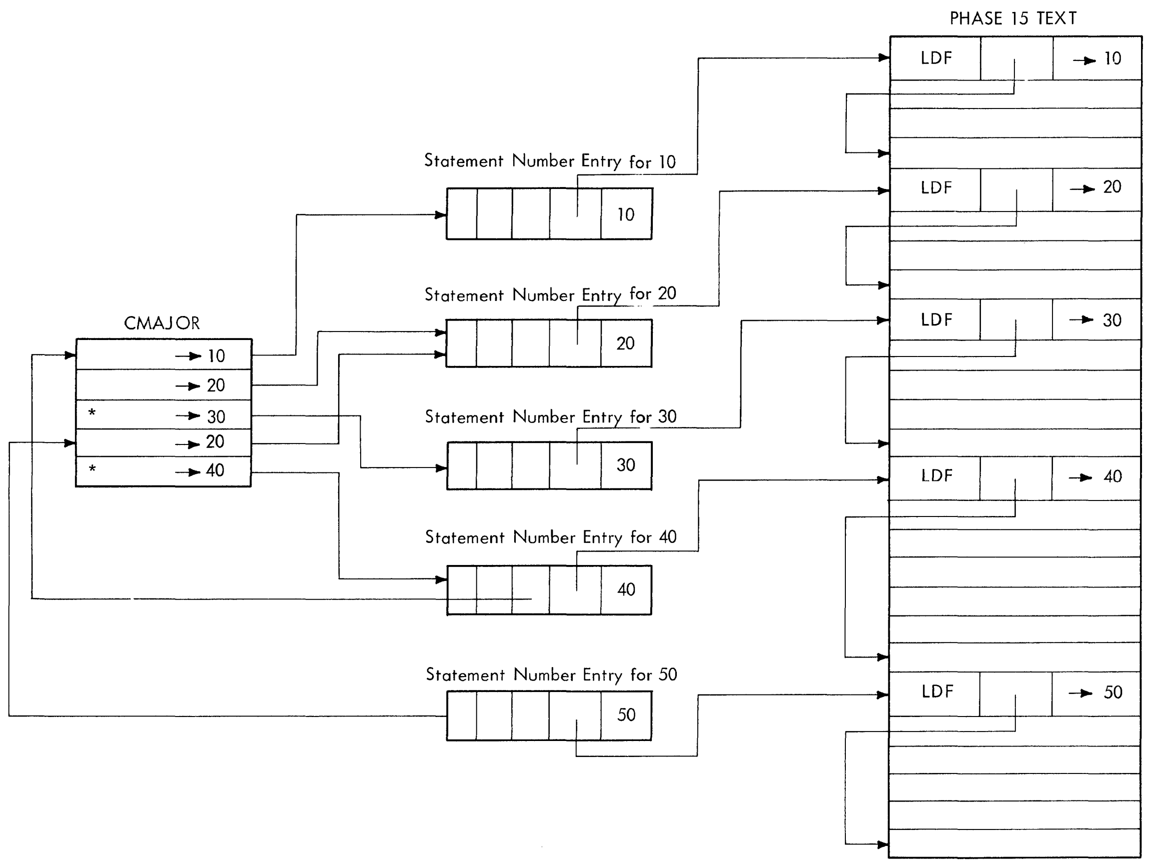

PHASE 15 TEXT
LDF
→ 10
LDF
→ 20
LDF
→ 30
LDF
→ 40
LDF
→ 50
Statement Number Entry for 10
10
Statement Number Entry for 20
20
Statement Number Entry for 30
30
Statement Number Entry for 40
40
Statement Number Entry for 50
50
CMAJOR
→ 10
→ 20
* → 30
→ 20
* → 40

Figure 7. Backward Connection Information

CORAL PROCESSING

CORAL, the last overlay segment of phase 15, performs five functions. It first converts phase 10 data text to a form more easily evaluated by phase 25. CORAL then assigns addresses relative to the start of an object module to all symbolic operands -- variables, constants, and arrays. During the assignment of relative addresses to variables, CORAL rechains the data text in order to simplify the generation of text card images by phase 25. CORAL assigns space in the address constant table (NADCON) for unknown references -- call-by-name variables, library routines, and namelist names. This reserved space will be filled by later phases. Lastly, as a user option, CORAL prints a storage map of named items -- variables, arrays, and external references -- as recorded in the information table. (Chart 09 shows the overall logic flow of CORAL).

Translation of Data Text

The first section of CORAL, subroutine NDATA, translates data text entries from their phase 10 format to a form more easily processed by phase 25. 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 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 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 characters 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 characters are processed in the same manner, they are described together.

Constants, Variables, and Hollerith Character Strings Used as Arguments: Subroutine CONST first assigns relative addresses to the constants of the module. Then, subroutine VARA assigns addresses to the variables and Hollerith character strings. (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, which is the initial value of the location counter. 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: Subroutine VARA 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 concepts of span is discussed in Appendix G.) 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.

VARA 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, VARA 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. VARA 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, VARA does not take any additional action before it processes the next array. If the location counter value exceeds 4095, VARA establishes a new address constant, resets the location counter, and processes the next array. After all arrays have relative addresses, VARA returns control to CORAL, which calls subroutine EQVAR 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 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 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 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 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 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 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 then processes the next element of the group.

If the desired address constant does not exist, EQVAR 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 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 computes the base value for the next group, if any. This base value is equal to the base value of the group just processed plus the size of that group. The next group is then processed.

Common Variables and Arrays: Subroutine COMVAR considers each common block of the source module, in turn, for relative address assignment. For each common block, COMVAR 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, COMVAR assigns each common block of the source module a relocatable origin of zero. It achieves the origin of zero by assigning to the first element of a common block a relative address consisting of an address constant and a displacement whose sum is zero. For example, both the address constant and the displacement for the first element in a block can be zero. Also, the address constant can be -16 and the displacement +16. Note that the address constant in the latter case is negative. Negative address constants are permitted, and may be a by-product of the assignment of addresses to common variables and arrays. They evolve from the manner in which the relative addresses are assigned to arrays. A relative address assigned to an array is equal to its actual relative address minus the span of that array. The actual relative address of each array in a common block is equal to the offset computed for

---

[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.

it during the common and equivalence processing of the first segment of phase 15, STALL. From the offset of each array in the common block under consideration, COMVAR 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, COMVAR 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, COMVAR 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, COMVAR 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.

<u>Variables and Arrays Equivalenced into Common:</u> Subroutine COMVAR processes variables and arrays that are equivalenced into common in much the same manner as EQVAR processes 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.

<u>Adcon and Base Variable Assignment:</u> 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 <u>base variables</u>. Only those operands receiving coordinate assignments are available for full register assignment during phase 20.

<u>Rechaining Data Text</u>

During the assignment of relative addresses to variables, subroutine DATACH rechains the data text entries. Their previous chaining (set by phase 10) was according to their order of appearance in the source program. DATACH 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.

<u>Reserving Space in the Adcon Table</u>

After relative address assignment is completed, subroutine EXTRNL reserves space in the adcon table for certain special references. It 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 EXTRNL if a particular reference belongs to one of these categories. For each special reference that EXTRNL detects, it reserves four bytes in the adcon table. Phase 25 places the needed address constants in the reserved spaces.

<u>Producing a Storage Map</u>

Lastly, as a user option, subroutine STMAP produces a storage map of named items. These items include variables, arrays, function or subroutine references, and statement functions (SF). For each of these, except function or subroutine references, the map contains the name, location, type, and tag. (The tag indicates whether a variable appeared in a COMMON or EQUIVALENCE statement or in both. It is set by phase 10 or by CORAL.) For a function or subroutine reference the map lists the name and whether the reference is external or in IFUNTB table.

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 no optimization, an intermediate amount of optimization, or complete optimization. Thus, the functions performed by phase 20 depend on the optimization specified for the compilation.

- If no optimization has been specified, phase 20 assigns to intermediate text entry operands the registers they will require during object module execution (this is called <u>basic register assignment</u>). 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 an intermediate amount of optimization has been specified, two processes are carried out:

  1. The first process, call <u>full register assignment</u>, 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 <u>branch optimization</u>, 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 complete optimization has been specified, other measures are taken to improve object-module efficiency. Complete 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. Complete optimization involves three general procedures:

1. The first, called <u>text optimization</u>, 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 intermediate 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 intermediate-optimized 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) 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 intermediate-optimization and complete-optimization. The third group of routines performs branch optimization and is also used in the paths for both intermediate-optimization and complete-optimization. The fourth group determines the structure of the source module and is used only in the path for complete-optimization. The fifth group performs loop selection and again is only executed in complete-optimization. The final group performs text optimization and is only used in complete-optimization.

The control routine governs the sequence of processing through phase 20. The processing sequence to be followed is determined from degree of optimization 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. Each block is passed by the control routine to the basic register assignment routines for processing. When all blocks are processed, the control routine passes control to the FSD, which calls phase 25.

When intermediate-optimization is specified, the control routine passes the entire module to the full register assignment routines and then to the routines that compute the size of each text block. When all block size information is gathered, the control routine calls the routine that computes, using the block size information, the displacements required for branching optimization. Control is then passed to the FSD.

When the control path for complete 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 routines that compute the size of each text block and then to the routine that computes, using the block size information, the displacements required for branching optimization. Control is then passed to the FSD.

REGISTER ASSIGNMENT

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

Each text entry has associated operand and base address status information that is set up by phase 20 in the status field of that text entry (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

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 2.

Table 2. 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 5) 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.

<u>Status Setting:</u> Subroutine SSTAT 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 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 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 uses a similar procedure to set the status of operand 3.

To set the status of the base address of operand 2, SSTAT determines the relationship of that base address to the current base address (see note). If they correspond, SSTAT 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 wise, it sets the status to indicate that it is in main storage.

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

<u>Note:</u> 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 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 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 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 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 assigns operand 2 a register. It selects the register according to the type of operand (refer to Table 2), 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 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 2, 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 does not assign a register to the base address of operand 2. SPLRA 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 assigns the appropriate register (refer to Table 2) to that operand, and enters the number of that register into the R3 field.

Operand 1 is always assigned a register. SPLRA selects the register according to the type of operand 1 (refer to Table 2), 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 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 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

During full register assignment, 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 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 first block (i.e., entry block) 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), table building (subroutine FWDPAS), local assignment (subroutine BKPAS), global assignment (subroutine GLOBAS), and text updating (subroutine STXTR). The control routine of phase 20 (LPSEL) passes control to the full register assignment control routine, 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) makes a pass over the MVD table and the dictionary entries for the variables and constants in the loop passes to it, and constructs the eminence table (EMIN) for the module, which indicates the availability of the variables for global assignment. The routine then calls the table-building routine to process the first block in the module.

2. The table-building routine (FWDPAS)

builds the required set of local assignment tables for the block and, at the same time, adds information to the global assignment tables under construction. It then passes control to the local assignment routine to process the block. When processing of the block is completed, control is returned to REGAS.

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

4. The control routine (REGAS) selects the next block in the module, and passes it to the table-building routine, which then passes control to the local assignment routine. This process continues until all blocks in the module have been processed by the table-building and local assignment routines.

5. The control routine passes control to the global assignment routine, which performs global assignment for the module.

6. When global assignment is complete, the control routine calls the text updating routine (STXTR) to complete register assignment by entering the results of global assignment into the text entries for the module. Control is then returned to the control routine of phase 20 (LPSEL).

Table Building for Register Assignment: The table-building routine 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 assignement tables can accommodate information for 100 text entries. If 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 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) consults the local assignment tables to determine that operand's eligibility. If the operand is eligible, BKPAS 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.

BKPAS 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 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 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 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, no common variables may be considered for local assignment and no real operands can 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 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), 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 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 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 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) 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 interrogates the completed global assignment table to determine if a global assignment has been made for the operand. If it has, STXTR enters the number of 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 records the number of 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 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 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 0 or 15) as the base register of the operand. STXTR 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 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 compatability 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 the full register assignment control routine (REGAS) upon completion of its functions. REGAS, in turn, returns control to the control routine of the phase (LPSEL).

BRANCHING OPTIMIZATION

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 calculates the sizes of all text blocks after full register assignment for the module is completed. Subroutine LYT then uses the gathered block size information to determine the blocks that can be branched to by means of RX-format branch instructions. BLS 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 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 and for each required prologue and epilogue, if a subprogram program is being compiled (refer to Phase 25, "Prologue and Epilogue Generation").

After BLS computes all block sizes, subroutine LYT 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 LYT 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 four is reserved. The registers are reserved, as needed, in the following order: register 13, 11, 10, and 9.

Note: Phase 20 also reserves register 12 to contain the relative address of the "constants" portion of text information (see Figure 12). It is used to refer to the constants and/or variables that occupy locations within the first 4096 bytes of the text information portion of the object module.

Reserved Register Addresses

The addresses placed into the reserved registers as a result of the execution of the initialization instructions (refer to Phase 25, "Initialization Instruction") are:

- Register 13 - address of main program (or subprogram) save area.[1]
- Register 11 (if reserved) - address of the save area plus 4096.
- Register 10 (if reserved) - address of the save area plus 2(4096).
- Register 9 (if reserved) - address of the save area 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 12), 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, subroutine LYT 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, LYT enters the displacement of that block (from the address) into the statement number entry for the statement number associated with the beginning of that block. 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 LYT 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 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 complete optimization, the structural determination routines of phase 20 (TOPO and BAKT) 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.[2]
- 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 <u>back dominator</u> 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]Register 13 is used to refer to the adcon table, which resides in text information immediately after the initialization instructions (see Figure 12).

[2]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 8 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.

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.

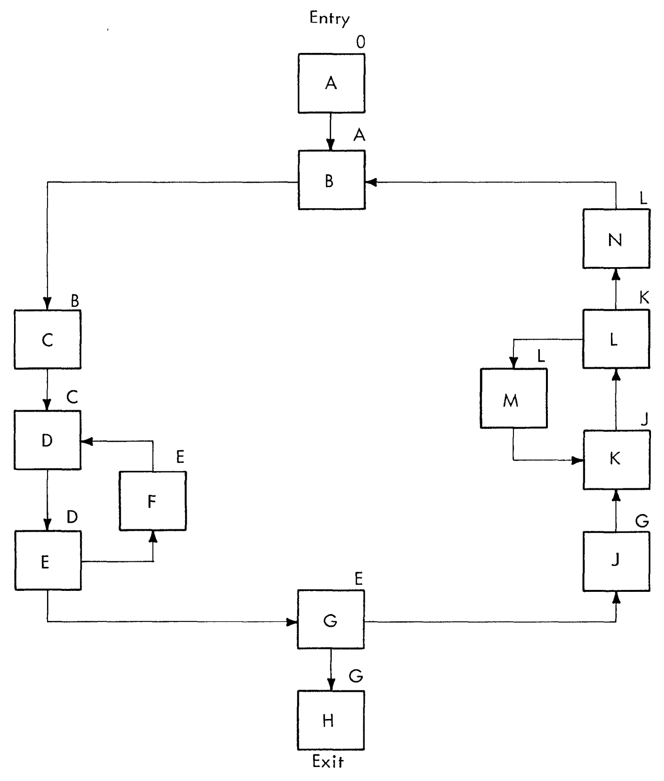


Figure 8. Back Dominators

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 9 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.

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.

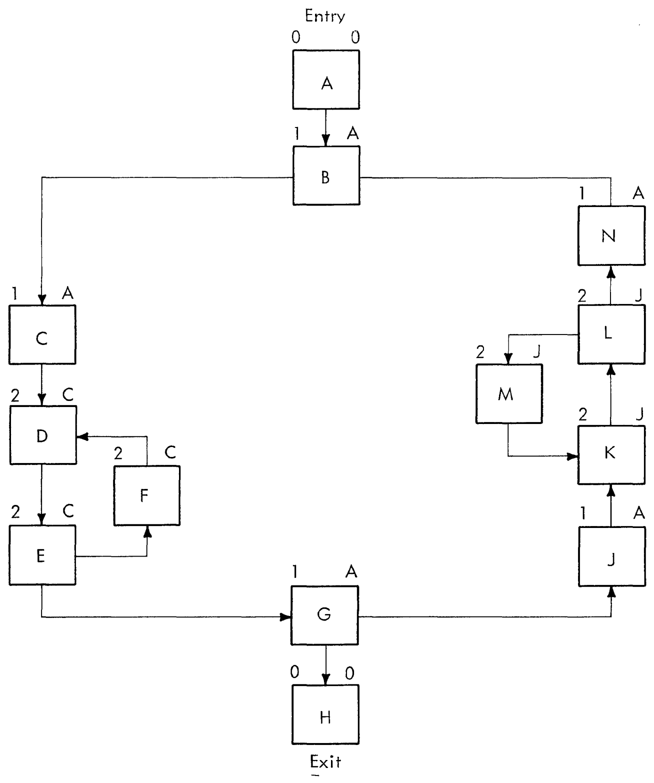


Figure 9. Back Targets and Depth Numbers

Determination of Back Dominators

Subroutine TOPO determines the back dominator of each text block by examining the connection information for that block. The first block processed by TOPO 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 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 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 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 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 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 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 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 then determines the actual back dominators of blocks that were assigned provisional back dominators.

For each block assigned a provisional back dominator, subroutine TOPO 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 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 determines the back target and depth number of each text block.

Determination of Back Targets and Depth Numbers

Subroutine BAKT 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 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 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 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 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 assigns that block as the back target of the block under consideration. BAKT 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 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 assigns its back target as the back target of the block under consideration. BAKT 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 obtains and examines the nearest block to block I in its chain of back dominators that has two or more forward connections. BAKT 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 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 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 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 obtains and examines the nearest block to block I in its chain of back dominators that receives control from two or more blocks. BAKT 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 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 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 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 assigns it both a back target and depth number of zero.

Identifying and Ordering Loops for Processing

Subroutine BAKT 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 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 complete optimization. (BAKT records the value 1 in the loop number field of the statement number entry for each block in that loop.) BAKT 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 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 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 During Complete Optimization"), 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. 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, 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 passed to the control routine of phase 20 (LPSEL).

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

After BIZX 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 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, 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 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 passes control to LPSEL, which calls the loop selection routines.

LOOP SELECTION

The loop selection routines of phase 20 (TARGET, BASVAR, and BSYONX) select the loop to be processed and provide 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 routines. The control routine of phase 20 (LPSEL) sets this parameter to one after the process of structural determination is complete. The loop selection routine TARGET 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, 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 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 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 uses the value of the loop number parameter as a comparand to select the loop to be processed. TARGET 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.
- 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 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 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 <u>single</u> block to which the loop passes control after its execution is complete.

To locate the forward target (if any), the loop selection routine BSYONX 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.) BSYONX 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 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 searches the statement number chain for the first and last entries having the current loop number. The block 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 BASVAR forms LMVS by combining, via or 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. BASVAR 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. BSYONX forms it during its search for the forward target of the loop. BSYONX 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. BSYONX 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

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 five logical sections: common expression elimination, forward movement, backward movement, strength reduction, and constant expression reordering.

- Common expression elimination optimizes the execution of a loop by eliminating unnecessary re-computations of identical arithmetic expressions.
- Forward movement optimizes the execution of a loop by relocating to the forward target computations essential to the module but not essential to the current loop.
- 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.
- Constant expression reordering optimizes the execution of a loop by reordering text entries involving the interaction of constants. The resultant text entry may be eliminated or may be relocated into the back target.
- 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 three of the above sections are similar in that they examine text entries in strict order of occurrence within the loop.

The last two sections do 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 3. 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.

Table 3. 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	Constant Expression Reordering
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 and Constant Expression Reordering
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 and Constant Expression Reordering
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 <u>IBM System/360 Operating System: FORTRAN IV</u>.

[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.

<u>Common Expression Elimination</u>

The object of common expression elimination, which is carried out by subroutine XPELIM, 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 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 attempts to locate a matching text entry. A text entry matches the candidate if operand 2, operand 3, and the operator of that text entry are identical to those of the candidate. If either operand 2 or 3 of the matching text entry is redefined between that text entry and the candidate, the match is not accepted. The search for the matching text entry takes place in the following locations:

• In the same block as the candidate,

between the first text entry and the candidate.

- In a back dominator (see note) of the block in which the candidate resides.

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

When a matching text entry is found, XPELIM 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 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 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 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 forward movement.

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

Forward Movement

Forward movement, which is carried out by subroutine FORMOV, optimizes a loop by moving text entries from the loop to the forward target of the loop, an area where they are executed less often. If the loop does not have a defined forward target, forward movement is bypassed and backward movement is initiated. Only text entries that are not required in the loop are moved during forward movement. An example of such a text entry is one whose operand 1 is not needed elsewhere in the loop. The following paragraphs describe the processing that occurs during forward movement.

Within the loop currently being optimized, FORMOV examines each uncompleted block in the chain of back dominators of the forward target (starting with the nearest back dominator of the forward target and proceeding as described in common expression elimination) for text entries that are candidates for forward movement. (The block is examined in a bottom-to-top fashion.) A text entry is a candidate for forward movement if:

- The text entry contains an arithmetic or logical operator.
- Operand 1 of the text entry is not used in another text entry in the loop.

When a candidate is found, FORMOV performs forward movement of the candidate in one of two ways:

- If the operands of the candidate are not defined in the text entries between candidate and the forward target, FORMOV moves the entire candidate to the beginning of the forward target.
- If an operand of the candidate is defined and if the expression (i.e., operand 2-operator-operand 3) in the candidate contains a variable and temporary, joined by a commutative operator, FORMOV generates a text entry to store the variable in a new temporary. It then replaces the candidate with this text entry, moves the candidate to the forward target, and replaces the variable with a reference to the new temporary.

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 that is also a back dominator of the forward target is selected and its text entries undergo forward movement. When all uncompleted blocks that are back dominators of the forward target and within the confines of 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 forward movement is illustrated in Chart 12. An example of forward movement is given in Appendix D.

Backward Movement

Backward movement, which is performed by subroutine BACMOV, 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 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 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 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 13. 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 integer constants.

Elimination of Simple Stores: BACMOV removes 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 paragraphs describe the processing that occurs during simple-store elimination.

During the scan of each uncompleted block for text entries to be moved to the back target, BACMOV checks for simple stores that are candidates for elimination. A simple store is a candidate for elimination if its operand 1 is a variable.

When a candidate is found, BACMOV examines the characteristics of its operands to determine if the candidate can be eliminated. The various combinations of operand characteristics that permit a candidate to be eliminated are given in table 4. If the characteristics of the operands of the candidate conform to any one of these ten combinations, BACMOV eliminates the candidate.

Table 4. Operand Characteristics That Permit Simple-Store Elimination

Operand 1 busy-on-exit from block	Operand 1 refined below in block	Operand 2 redefined below in block	Operand 1 used in block below redefinition of operand 2	Operand 1 re-defined below before redef-inition of operand 2	Operand 1 redefined be-low between redefini-tion of operand 2 and first use of operand 1 that follows redefini-tion of operand 2
1. No	No	No	X	X	X
2. No	Yes	No	X	X	X
3. Yes	Yes	No	X	X	X
4. No	No	Yes	No	X	X
5. No	Yes	Yes	No	Z	X
6. No	Yes	Yes	Yes	Yes	X
7. No	Yes	Yes	Yes	No	Yes
8. Yes	Yes	Yes	No	Z	X
9. Yes	Yes	Yes	Yes	Yes	X
10. Yes	Yes	Yes	Yes	No	Yes

X = condition cannot exist because of previous characteristics of operands.
Z = characteristic is irrelevant.

It does this by replacing the uses of operand 1 (of the candidate to be eliminated) 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 (i.e., if a redefinition of either operand does not occur).

BACMOV then deletes the candidate. An example of simple-store elimination is illustrated in Appendix D.

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 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 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.

Constant Expression Reordering

Constant expression reordering, which is performed by subroutine AGGLUT, optimizes the loop being processed by reducing the number of calculations that must be performed within the loop to evaluate arithmetic operations involving constants. For example, assume that the arithmetic operation A/3.0*4.0, represented by the pair of text entries T=A/3.0 and T1=T*4.0, appears within a loop. The number of calculations that must be performed within the loop to evaluate this operation can be reduced by dividing 4.0 by 3.0 outside the loop and inserting the result back into the loop in such a fashion that the operation is reordered and simplified to A*T2 (where T2 equals the result of 4.0 divided by 3.0). The resultant text for the above operation would appear as T1=A*T2, which remains in the loop, and T2=4.0/3.0, which is performed outside the loop.

The following paragraphs discuss the processing that occurs during constant expression reordering.

Within the loop currently being processed, AGGLUT examines each uncompleted block for pairs of text entries that are candidates for constant reordering. A pair of text entries to be a candidate must meet all of the following conditions:

- Both text entries have arithmetic operators.
- The expressions in both text entries are functions of a variable (or temporary) and a real constant (type 5 or type 6 text entries).
- Operand 1 of both text entries is a temporary.
- The text entries have a common temporary that is defined in one text entry and used in the other.

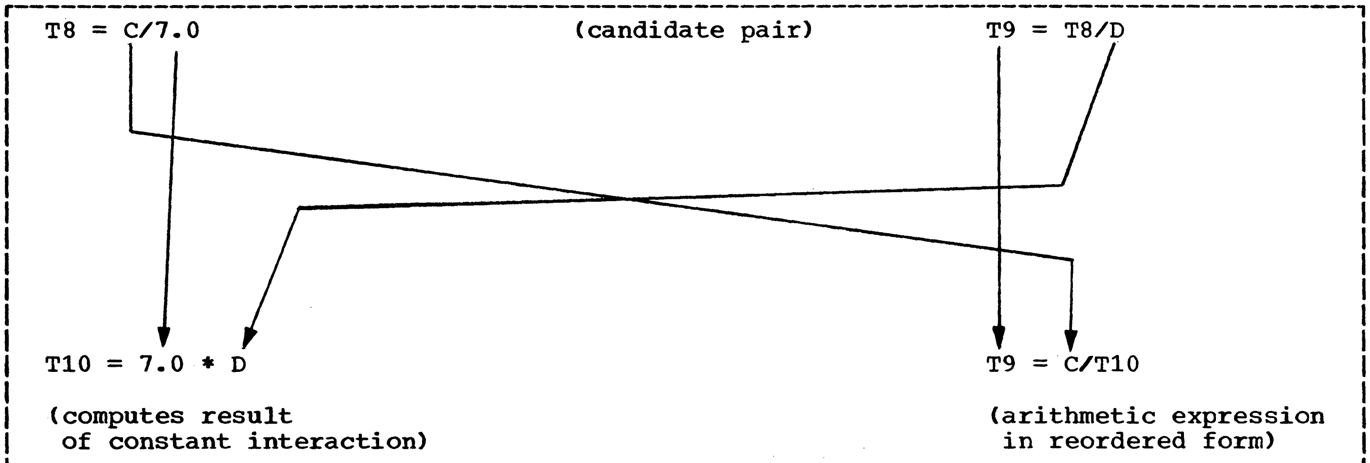
Note: The text entry in which the common temporary is defined must precede the text entry in which it is used.

A pair of text entries with these characteristics represents an arithmetic operation that may be reordered and simplified by means of transformations and an operator table.

The transformations indicate the operand movement required to reorder the expression represented by the pair of candidate text entries. There are two transformations:

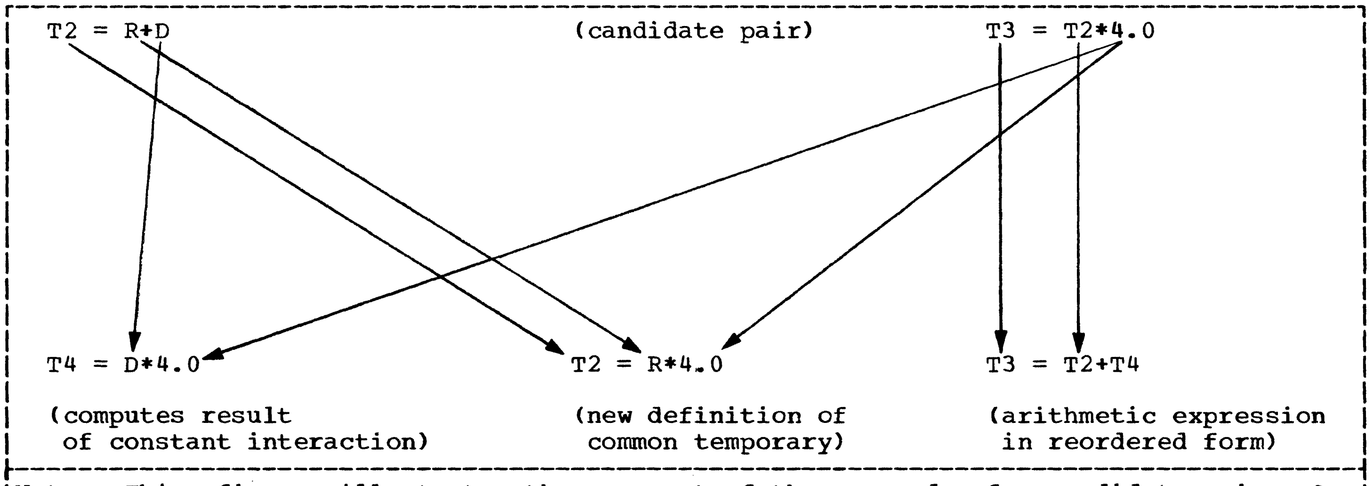
1. One is applied to candidate pairs when the text entries for both have either multiplicative or additive operators. The application of this transformation (see Figure 10) reorders the operation represented by the candidate pair and simplifies its calculation by eliminating a text entry. (The text entry eliminated is the text entry of the candidate pair in which the common temporary is defined.)
2. The second transformation is applied to candidate pairs, one text entry of which has an additive operator (see note), and the other of which has a multiplicative operator. The application of this transformation (see Figure 11) reorders the arithmetic expression represented by the candidate pair and generates additive constants, which may be subsequently used to eliminate text entries.

Note: The text entry in which the common temporary is defined must have the additive operator.


T8 = C/7.0
(candidate pair)
T9 = T8/D
T10 = 7.0 * D
T9 = C/T10
(computes result of constant interaction)
(arithmetic expression in reordered form)

Note: This figure illustrates the movement of the operands of a candidate pair, the text entries of which both have multiplicative operators. The operators in the resultant text entries, T10=7.0*D and T9=C/T10, are obtained from the operator table. The text entry T10=7.0*D which computes the result of the interaction of the constants, is placed into the back target. (In this application; D is assumed to be a stored constant.)

Figure 10. Multiplicative-Multiplicative or Additive-Additive Transformation


T2 = R+D
(candidate pair)
T3 = T2*4.0
T4 = D*4.0
T2 = R*4.0
T3 = T2+T4
(computes result of constant interaction)
(new definition of common temporary)
(arithmetic expression in reordered form)

Note: This figure illustrates the movement of the operands of a candidate pair. One text entry contains an additive operator; the other contains a multiplicative operator. The operators in the resultant text entries are obtained from the operator table. The text entry T4=D*4.0, which computes the result of the interaction of the constants, is placed in the back target. (In this application, D is assumed to be a stored constant.)

Figure 11. Additive-Multiplicative Transformation

The operator table (refer to Appendix A, "Operator Table") indicates the operators that are required to reorder the arithmetic operation represented by a pair of candidate text entires. Arguments to the table are, respectively:

- The operator of the text entry in which the common temporary is defined.
- The operator of the text entry in which the common temporary is used.

The operators obtained from the table are, respectively:

- The operator of the text entry used to combine the constants.
- The operator of the text entry representing a new definition of the common temporary if any is required.
- The operator of the text entry that represents the arithmetic operation in reordered form.

Note: If the operators of the candidate pair are either both multiplicative or both additive, a new definition of the common variable is not required to reorder the arithmetic operation.

Use of Transformations and Operator Table: Subroutine AGGLUT uses the transformations and the operator table in combination to determine the text entries that are required to reorder the arithmetic operation represented by the candidate pair. It determines the operands of the required text entries from the appropriate transformation, which is selected according to the nature of the operators of the candidate pair. It determines the operators of the required text entries by matching the operators of the candidate pair to the operators in the argument fields of the entries in the operator table. When the entry whose arguments match the operators in the candidate pair is found, AGGLUT obtains the functions (i.e., the operators to be used in the required text entries) of that entry and uses them as the operators of the required text entries.

Residence of Text Entries After Reordering: The text entries that result from the processing carried out by subroutine AGGLUT on a candidate pair occupy the following locations:

- The text entry that computes the result of the interactions of the constants resides in the back target of the loop (see note).
- The text entry representing a new definition of the common temporary, if any such text entry is required, replaces the text entry of the candidate pair in which the common temporary was defined.
- The text entry representing the arithmetic operation in reordered form replaces the text entry of the candidate pair in which the common temporary was used.

Note: This text entry does not exist if the interacting constants are both absolute. Prior to placing the resultant text entries into their appropriate locations, AGGLUT determines if both interacting constants are absolute. If they are, it computes the result of their interactions and constructs a dictionary entry for the result. The result is used as the appropriate used operand of the text entry that represents the arithmetic expression in reordered form.

Processing Procedure: The left-most column of the operator table is divided into three groups:

- Group A--Multiplicative-multiplicative.
- Group B--Additive-Multiplicative.
- Group C--Additive-Additive.

The processing performed during constant expression reordering follows the order of these groups. AGGLUT first processes candidate pairs whose text entries both have multiplicative operators, (i.e., pairs of type 5 text entries); it next processes candidate pairs, one text entry of which has an additive operator (a type 6 text entry) and the other a multiplicative operator (a type 5 text entry); and then processes candidate pairs whose text entries both have additive operators (pairs of type 6 text entries).

During constant expression reordering, AGGLUT first attempts to locate pairs of type 5 (group A) text entries that are candidates. If it finds any, it processes them. AGGLUT then attempts to locate pairs of text entries one of which is type 6 and the other type 5 (group B) that are candidates. If it locates any such pairs, it processes them. (If any group B candidate pairs are found, group A processing is repeated.) Finally, AGGLUT attempts to locate pairs of type 6 (group C) text entries that are candidates. If it finds any, it processes them.

All the candidate pairs 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 constant expression reordering. When all uncompleted blocks in the loop are processed, control is returned to the control routine, which passes control to the portion of phase 20 that continues text optimization through strength reduction.

The overall logic of constant expression reordering is illustrated in Chart 14. An example of this process is presented in Appendix D.

Additional Processing: After all type 6 candidate pairs have been processed, an additional process, the elimination of a type 6 text entry, is carried out if operand 1 of any of the remaining type 6 text entries is the index value of a subscripted variable (e.g., X(s T4, where T4 corresponds to operand 1 of a type 6 text entry). If such is the case, the index value of the subscripted variable is a function of a variable (or temporary) and an additive constant. (Consider that the index value is defined as T4=T2+K, which is a type 6 text entry.) Subroutine AGGLUT renders the type 6 text entry unnecessary and eliminates it by:

- Reducing the index value of the subscripted variable by the amount of the additive constant that appears in the type 6 text entry whose operand 1 corresponds to the index value.
- Increasing (by the above amount) either of the two elements (displacement or address constant) that combines with the index value to yield the address of the subscripted variable.

Note: The address of a subscripted variable is equal to the sum of (1) the index value computed from the subscript parameters, (2) the displacement assigned to the array containing the subscripted variable, and (3) the address constant (base address) assigned to the array containing the subscripted variable.

AGGLUT reduces the index value of the subscripted variable by the amount of the additive constant by replacing the index value with the used variable (or temporary) of the type 6 text entry whose operand 1 corresponds to the index value.

Whether constant expression reordering increases the displacement or the address constant depends upon the nature of the additive constant.

If the additive constant is an absolute constant and its magnitude is such that, when it is added to the displacement assigned to the array containing the subscripted variable, the result is less than 4096, AGGLUT incorporates the additive constant into the displacement. It accomplishes this by adding the additive constant to the contents of the DP field of the subscript text entry (refer to Appendix A, "Phase 15 Intermediate Text Modifications"). (When phase 25 generates machine code for the subscript text entry, it adds the contents of the DP field to the displacement assigned to the array that contains the subscripted variable.)

If the additive constant is either an absolute constant, whose magnitude is such that the 4096 restriction is violated, or a stored constant, AGGLUT incorporates the additive constant into an address constant.

It does this by:

- Creating a new variable and replacing the subscripted variable with the new variable.
- Constructing a dictionary entry for the new variable and assigning it an address constant.
- Generating a text entry, the function of which is to insert into the address constant assigned to the new variable the sum of the value of the address constant assigned to the array containing the replaced subscript variable and the additive constant.
- Placing the generated text entry into the back target of the loop.

In either case, the address that results from combining the index value, the displacement, and the address constant (associated with the subscript text entry that results from constant expression reordering) is equivalent to the address that would result from combining the index value, the displacement, and the address constant associated with the original subscript text entry, where that text entry left unchanged.

Strength Reduction

Strength reduction, which is performed by subroutine REDUCE, 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 of lower frequency of execution. The processing of strength reduction 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 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 logic IF statement that controls the loop.

If the loop is a candidate, REDUCE 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:
   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.

---

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

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, IF (J≤3) where J is the branch variable and 3 is the branch constant.

2. If either of the constants in the inert text entry or the multiplicative text entry is a stored constant, REDUCE 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 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[2] (type 6).

If the loop is a candidate, the processing performed by REDUCE 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 15. An example showing both methods of strength reduction is given in Appendix D.

FULL REGISTER ASSIGNMENT DURING COMPLETE OPTIMIZATION

During complete optimization, full register assignment is carried out on module loops, rather than on the entire module, as is the case for intermediate 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 will approach a minimum. The processing of a loop by full register assignment differs from its 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"). Global assignment for a loop is described in the following text.

When processing a loop, the global assignment routine (GLOBAS) 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 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

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

is currently available. If that register is available, GLOBAS 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 therefore assigns the item an available register different from that assigned to it in the previously processed loop. GLOBAS 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 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 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 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 records all global assignments made for the current loop for use in the subsequent updating scan (see "Full Register Assignment") and also for incorporation, wherever possible, into subsequently processed loops.

BRANCHING OPTIMIZATION DURING COMPLETE OPTIMIZATION

During complete optimization, branching optimization is carried out in the same manner as during intermediate optimization. After all loops have undergone full register assignment, BLS is given control to calculate the size of each block. When the sizes of all blocks have been calculated, subroutine LYT 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 produces 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 an assembler language listing of the instructions and data of the object module. 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 instrucitons 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 IEM 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 12 shows the layout of storage that Phase 25 assumes in setting up text information.

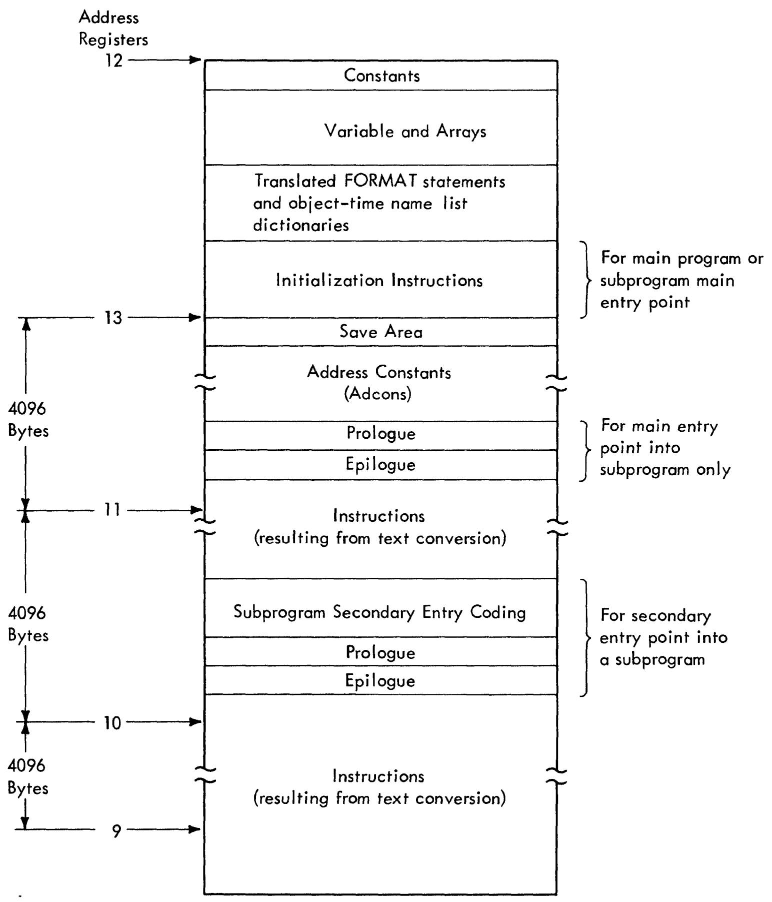

Address Registers
12
Constants
Variable and Arrays
Translated FORMAT statements and object-time name list dictionaries
Initialization Instructions
For main program or subprogram main entry point
13
Save Area
Address Constants (Adcons)
4096 Bytes
Prologue
Epilogue
For main entry point into subprogram only
11
Instructions (resulting from text conversion)
Subprogram Secondary Entry Coding
4096 Bytes
Prologue
Epilogue
For secondary entry point into a subprogram
10
4096 Bytes
Instructions (resulting from text conversion)
9

Figure 12. Storage Layout for Text Information Construction

Phase 25 constructs text information by:

- Reserving adcon table entries for the referenced statement numbers of the module.
- Entering the constants of the source module into TXT records.
- Reserving storage within text information for the variables and arrays of the module.
- Translating FORMAT statements (i.e., phase 10 format text) to a form recognizable by IHCFCOMH and entering the translated statements into TXT records. (IHCFCOMH, a member of the operating system library (SYS1.FORTLIB), performs object-time implementation of I/O statements. IHCFCOMH is explained in Appendix E.)
- Converting NAMELIST statements (i.e., phase 10 namelist text) to object-time namelist dictionaries, which are used by IHCFCOMH to implement READ-WRITE statements using NAMELIST statements.
- Generating the main program or subprogram initialization instructions and entering them into TXT records.
- Completing the processing of the adcon table entries and entering the resultant entries into TXT records.
- Assigning the initial values, as specified, to the variables and arrays appearing in phase 15 data text.
- Generating the prologue and epilogue instructions for a subprogram 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 21 shows the logic of phase 25 processing, down to, but not including, conversion of text to machine code.

Adcon Table Entry Reservation

Prior to beginning its construction of text information, subroutine LYT1 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 LYT1 scans the chain of statement number entries in the statement number/array table. For each encountered statement number that is referenced, LYT1 inserts into the appropriate field of the associated statement number entry a pointer to the next available entry in the adcon table. The actual value to be placed into the address constant set aside for a statement number is determined during text conversion (a subsequent phase 25 process), when the text representation of that statement number is encountered.

Note: If branching optimization is being implemented, LYT1 only reserves address constants 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, address constants are likewise reserved for the statement numbers appearing in computed GO TO statements. LYT1 scans the branch table chain (refer to Appendix A, "Branch Table"), and sets aside an entry in the ADCON table for each statement number for which a branch table entry was constructed. It also records a pointer to the address constant reserved for each fall through statement number in the initial branch table entry for that statement number. LYT1 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.

Constant Processing

Subroutine INITIL obtains the constants of the source module from their information table entries and places them into text information via TXT records. The address field of each such record specifies relative addresses for the constants that correspond to the relative addresses assigned to them by CORAL in Phase 15.

Variable and Array Processing:

Subroutine INITIL 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, INITIL 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.

FORMAT Statement Processing

If the source module contains READ/WRITE statements requiring FORMAT statements, the associate phase 10 format text must be put into a form recognizable by IHCFCOMH. Subroutine FORMAT 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 5. FORMAT 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.

NAMELIST Statement Processing

If the source module contains READ/WRITE statements using NAMELIST statements, subroutine NLIST 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,

Table 5. 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.

the relative address of its first element, and the information needed to locate a particular element of the array. NLIST obtains the above information, excluding the array name, from the information table.

NLIST 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.

Initialization Instructions

Phase 25 generates 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, which is generated by subroutine ATTACH.
- Subprogram main entry coding, which is generated by subroutine SUBR.
- Subprogram secondary entry coding, which is generated by subroutine ENTRY.

Once generated, these instructions are entered into TXT records.

Main Program Entry Coding: The initialization instructions generated by subroutine ATTACH 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 12288 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.
- 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.
- 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.

Subprogram Main Entry Coding: The initialization instructions generated by subroutine SUBR 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 "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, phase 25 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 ENTRY 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 (see Figure 12). 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 by subroutine ENTRY when the text representation of an ENTRY statement is encountered during the processing of intermediate text. These instructions reside in the "Instructions" section of text information.

Adcon Table Processing

Entries in the compile-time adcon table consist of the true address constants (base addresses) assigned by CORAL for local constants and variables and for common variables, pointers to information table entries for arguments and external reference address constants, temporaries and constants generated by phase 20, and reserved address constants, which are set aside for statement numbers. The output that the phase 25 subroutine NADOUT generates for the object-time adcon table consists of TXT records and RLD records in the case of true address constants. The RLD records provide the information needed to relocate the true address constants. (A type 5 ESD is output for each common block.) For argument address constants, NADOUT obtains the relative addresses of the arguments from their information table entries and places them into TXT records. It also includes RLD records for them. For an external reference address constant, NADOUT also includes a type 2 ESD record in addition to the TXT and RLD records. NADOUT outputs temporaries and generated constants in TXT records. It does not accompany them with RLD records.

NADOUT does not process address constants for statement numbers and for statement numbers appearing in computed GO TO statements at this time. However, it reserves storage for them within the "address constants" section of text information. It does this by incrementing the location counter by the number of address constants set aside for such items times four. The value of the updated location counter is then assigned as the relative address of the "prologue" if a subprogram is being compiled or of the "instructions" if a main program is being compiled.

As previously stated, the values to be placed into the address constants for statement numbers and statement numbers in computed GO TO statements are determined during text conversion, when that process encounters the END statement.

Phase 15 Data Text Processing

The phase 25 subroutine DATOUT 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 the 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.

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) 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 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) is a series of instructions that (1) return to the calling routine the values of "call by value" parameters (if any), (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 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.)

Residence of Prologues and Epilogues: The prologues and epilogues for secondary entry points do not reside in the "Prologue and Epilogue" section of text information (see Figure 12). This section is reserved for the prologue and epilogue of the main entry point. The prologue and epilogue for a secondary entry point into a subprogram are generated immediately after the secondary entry coding for the secondary entry point, and reside in the "Instructions" section of the text information following the secondary entry coding.

Text Conversion

The final function of phase 25 is the conversion of intermediate text into Operating System/360 machine code. (The text conversion process is controlled by subroutine MAINGN.) 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 seven processing paths to convert the text entry.

The seven processing paths are:

- Statement Number Processing.
- ENTRY Statement 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 22.

STATEMENT NUMBER PROCESSING: When the operator of the text entry indicates a statement number, MAINGN passes control to subroutine LABEL. LABEL then inserts the current value of the location counter, which is the relative address of the statement associated with the statement number, into the address constant for the statement number. When the associated statement is converted to machine code and placed into text information, it resides at an address equal to the value placed into the address constant. All branches to that statement are effected through the use of the address constant.

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 address constant for the statement number, subroutine LABEL determines if that statement number appears in a computed GO TO statement. If it does, LABEL 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 "Adcon Table Processing") when the text representation of the END statement is encountered.

ENTRY STATEMENT PROCESSING: When the operator of an intermediate text entry indicates an ENTRY statement, subroutine MAINGN passes control to subroutines ENTRY, PROLOG, and EPILOG. These subroutines generate the following for the subprogram secondary entry point:

- Subprogram secondary entry coding (refer to the section "Initialization Instructions").
- Prologue and epilogue (refer to "Prologue and Epilogue Generation").

The machine code instructions that constitute the above are entered into TXT records.

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 passes control to subroutine IOSUB, 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 enters it into TXT records.

CALL STATEMENT PROCESSING: When the operator of the text entry indicates a CALL statement, MAINGN passes control to subroutine CALLER 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. CALLER 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. The bit strip array for a particular operator contains strips of bits. 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 containing the 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.)

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

---

[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.

text information and by generating loader END loader record.

Subroutine END enters the address constant (i.e., relative address) for each statement number and for each statement number in a computed GO TO statement into a TXT record. The address inserted into each such record places the address constant into the storage reserved for it during ADCON table processing.

The loader END record must be the last record of the object module. Its functions are to signal the end of the object module and to inform the linkage editor of the size (in bytes) of the control section and the address of the main entry point of the control section.

EXTERNAL SYMBOL DICTIONARY

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 phase 25 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 phase 25 are:

- ESD-0 This is a section definition record for the source module being compiled.
- ESD-1 This record defines an entry point for the source module being compiled.
- ESD-2 This record is generated for an external subprogram name.
- ESD-5 This 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 <u>IBM System/360 Operating System: Linkage Editor, Program Logic Manual</u>.

RELOCATION DICTIONARY

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 phase 25 for each address constant 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 control section.

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

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 23. As errors are encountered by these phases, error table entries are created and placed into an error table. Each such entry consists of an internal statement number (i.e., a compiler generated number assigned to each source statement for identification purposes) for the statement that is in error, and 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:

- The internal 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.

Having constructed the parameter list, phase 30 calls subroutine MSGWRT, 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 Programmer's Guide for explanations of all the messages capable of being generated by the compiler.) 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

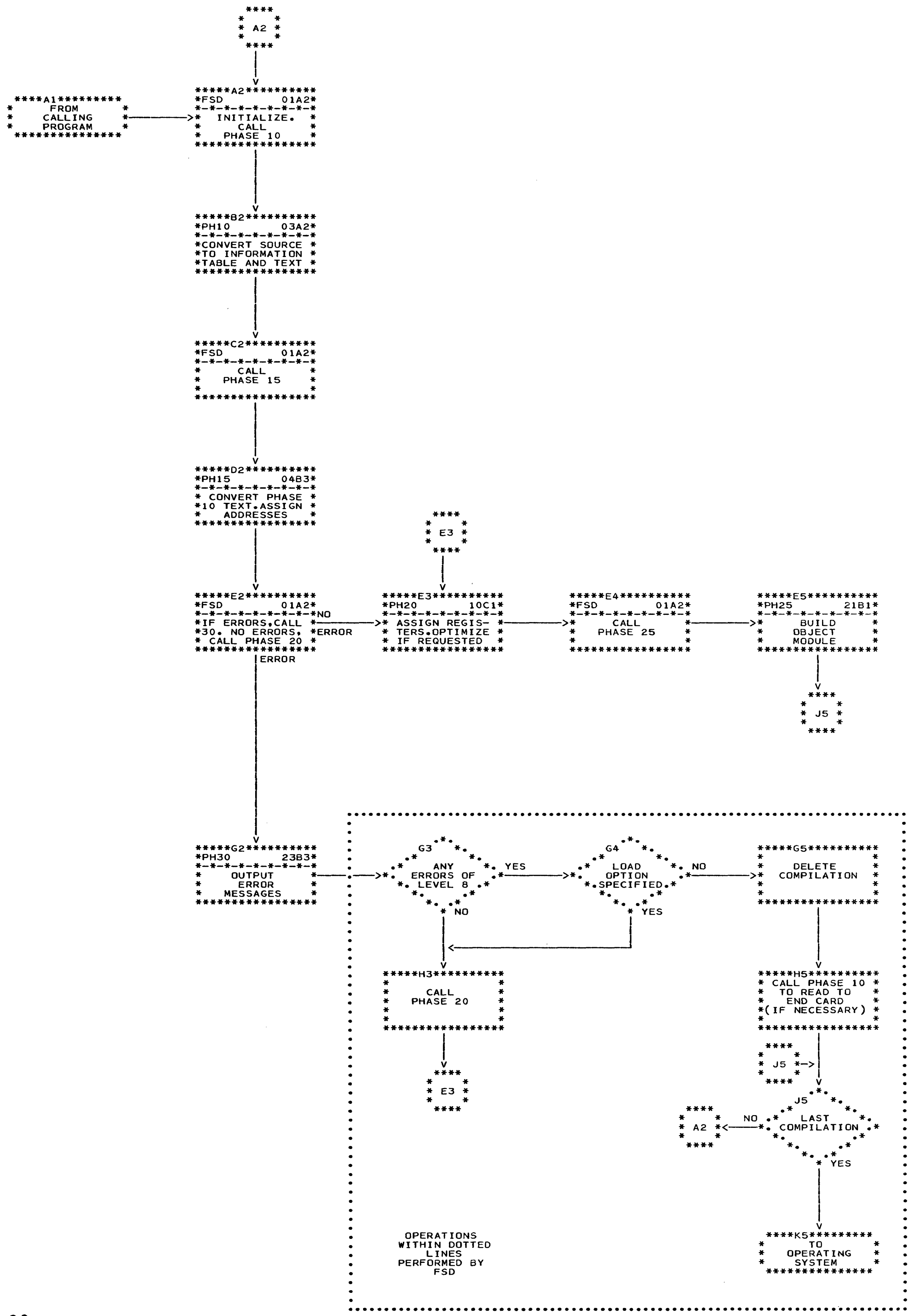


Chart 01. FSD Overall Logic

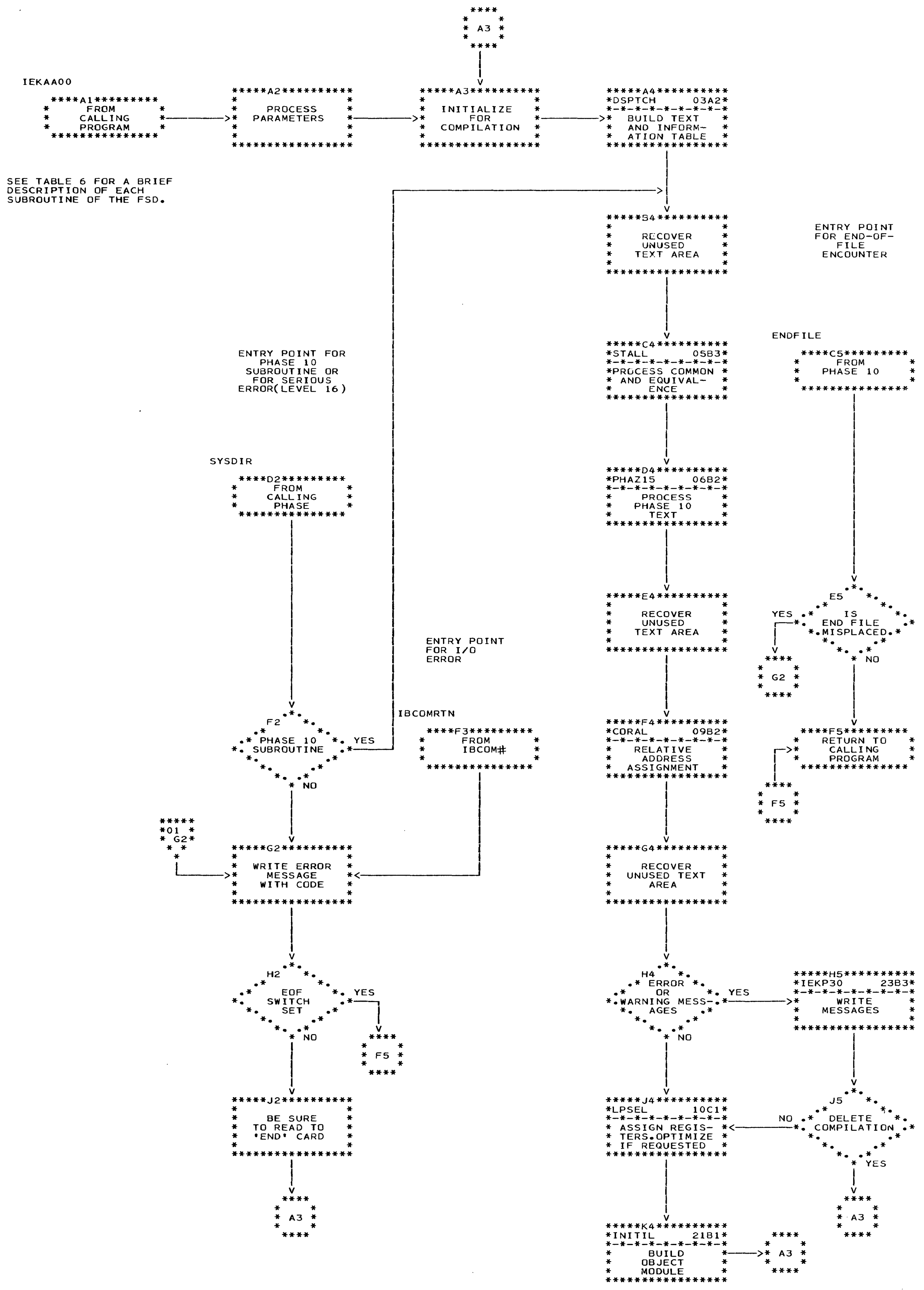

A3
IEKAA00
A1 FROM CALLING PROGRAM
A2 PROCESS PARAMETERS
A3 INITIALIZE FOR COMPILATION
A4 DSPTCH 03A2 BUILD TEXT AND INFORM- ATION TABLE
SEE TABLE 6 FOR A BRIEF DESCRIPTION OF EACH SUBROUTINE OF THE FSD.
B4 RECOVER UNUSED TEXT AREA
ENTRY POINT FOR END-OF- FILE ENCOUNTER
ENDFILE
C4 STALL 05B3 PROCESS COMMON AND EQUIVAL- ENCE
C5 FROM PHASE 10
ENTRY POINT FOR PHASE 10 SUBROUTINE OR FOR SERIOUS ERROR(LEVEL 16)
SYSDIR
D2 FROM CALLING PHASE
D4 PHAZ15 06B2 PROCESS PHASE 10 TEXT
E4 RECOVER UNUSED TEXT AREA
E5 IS END FILE MISPLACED
YES
NO
G2
ENTRY POINT FOR I/O ERROR
IBCOMRTN
F2 PHASE 10 SUBROUTINE
YES
NO
F3 FROM IBCOM#
F4 CORAL 09B2 RELATIVE ADDRESS ASSIGNMENT
F5 RETURN TO CALLING PROGRAM
F5
01 G2
G2 WRITE ERROR MESSAGE WITH CODE
G4 RECOVER UNUSED TEXT AREA
H2 EOF SWITCH SET
YES
NO
F5
H4 ERROR OR WARNING MESS- AGES
YES
NO
H5 IEKP30 23B3 WRITE MESSAGES
J2 BE SURE TO READ TO 'END' CARD
J4 LPSEL 10C1 ASSIGN REGIS- TERS.OPTIMIZE IF REQUESTED
J5 DELETE COMPILATION
NO
YES
A3
A3
K4 INITIL 21B1 BUILD OBJECT MODULE
A3

Chart 02. FSD Storage Distribution

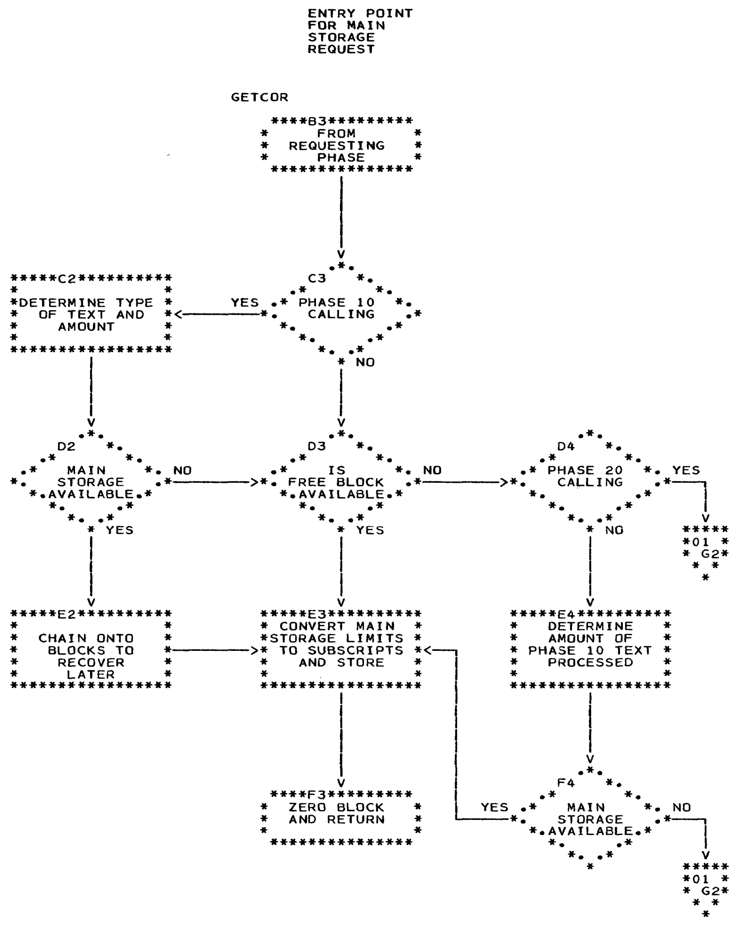

ENTRY POINT FOR MAIN STORAGE REQUEST
GETCOR
B3 FROM REQUESTING PHASE
C3 PHASE 10 CALLING
YES
NO
C2 DETERMINE TYPE OF TEXT AND AMOUNT
D2 MAIN STORAGE AVAILABLE
NO
YES
D3 IS FREE BLOCK AVAILABLE
NO
YES
D4 PHASE 20 CALLING
YES
NO
01 G2
E2 CHAIN ONTO BLOCKS TO RECOVER LATER
E3 CONVERT MAIN STORAGE LIMITS TO SUBSCRIPTS AND STORE
E4 DETERMINE AMOUNT OF PHASE 10 TEXT PROCESSED
F3 ZERO BLOCK AND RETURN
F4 MAIN STORAGE AVAILABLE
YES
NO
01 G2

Table 6. FSD Subroutine Directory

Subroutine	Function
AFIXPI	Exponentiation of integers by integers.
AFRXPI	Exponentiation of reals by integers.
GETCOR	Allocates and keeps track of main storage used in the construction of the information table and for collecting text entries.
IEKAA00	Initializes compiler processing and calls the phases for execution.
IEKFCOMH	Controls compile-time I/O. (Corresponds to IHCFCOMH; refer to Appendix E.)
IEKFIOCS	Interface between IEKFCOMH and BSAM. (Corresponds to IHCFIOSH; refer to Appendix F.)
IEKUATPT	Unit assignment table for IEKFIOCS.
IHCFMAXI	Maximizing service routine for integers.
IHCFMAXR	Maximizing service routine for reals.
SYSDIR	Deletes compilation if requested.
SYSTAB	Dumps internal text and tables.
SYSTRC	Diagnostic trace routine.

Chart 03. Phase 10 Overall Logic

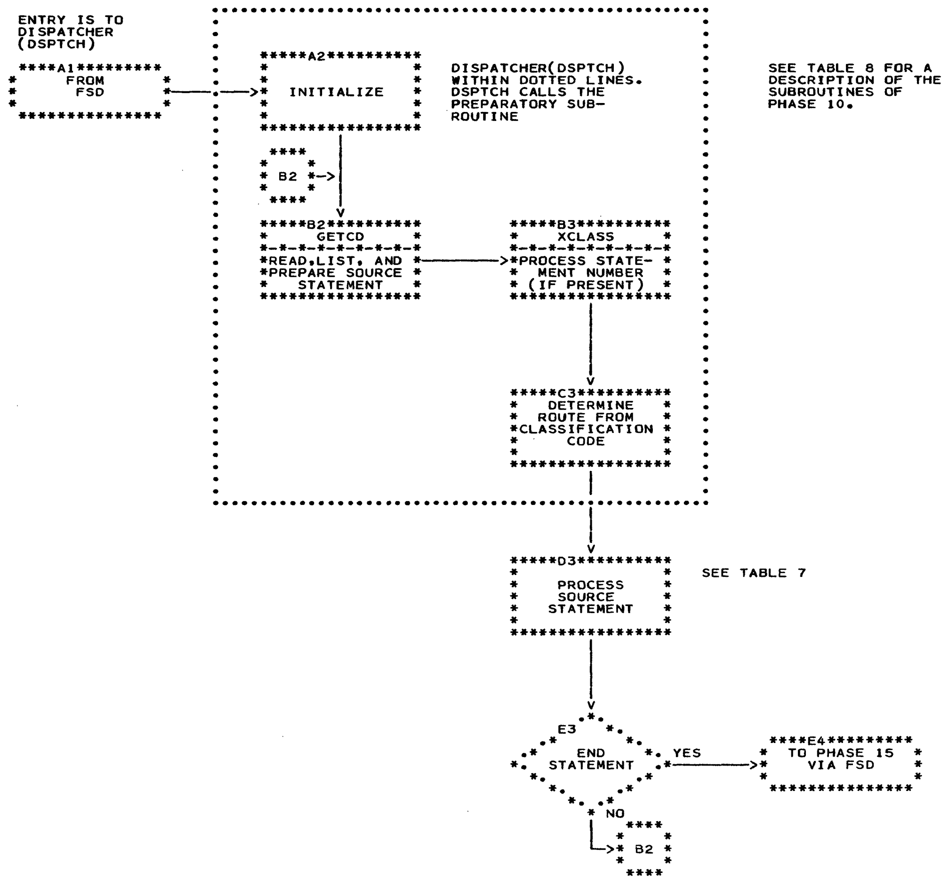

ENTRY IS TO DISPATCHER (DSPTCH)
A1 FROM FSD
A2 INITIALIZE
DISPATCHER(DSPTCH) WITHIN DOTTED LINES. DSPTCH CALLS THE PREPARATORY SUB-ROUTINE
SEE TABLE 8 FOR A DESCRIPTION OF THE SUBROUTINES OF PHASE 10.
B2
B2 GETCD READ,LIST, AND PREPARE SOURCE STATEMENT
B3 XCLASS PROCESS STATE-MENT NUMBER (IF PRESENT)
C3 DETERMINE ROUTE FROM CLASSIFICATION CODE
D3 PROCESS SOURCE STATEMENT
SEE TABLE 7
E3 END STATEMENT
YES
E4 TO PHASE 15 VIA FSD
NO
B2

Table 7. Phase 10 Source Statement Processing

Statement Type	Main Processing Subroutine	Subroutines Used
ARITHMETIC	XARITH	COMAST, GRPKEQ, MINSLS, PRELOG, RTPRQT, TXTBLD[1]
STATEMENT FUNCTION	XASF/XASF2	GETWD, ERROR, PUTX, CSORN, SYMTLU
DIMENSION	XDIM	GETWD, CSORN, ERROR, SYMTLU
EQUIVALENCE	XEQUI	GETWD, SYMTLU, ERROR, LITCON
COMMON	XCOMON	GETWD, SYMTLU, ERROR
EXTERNAL	XEXT	GETWD, ERROR, SYMTLU
TYPE (INTEGER, REAL, ETC.)	XTYPE	GETWD, ERROR, SYMTLU, PUTX
DO	XDO	GETWD, ERROR, LITCON, SYMTLU, PUTX, CDOPAR
SUBROUTINE, CALL ENTRY, FUNCTION	XSUBPG	GETWD, ERROR, SYMTLU, PUTX
READ, WRITE, PRINT, PUNCH	XIOOP	GETWD, ERROR, CSORN, PUTX, LITCON
NAMELIST	XNMLST	GETWD, SYMTLU, PUTX, ERROR
BACKSPACE, REWIND, END FILE	XBCKRW	GETWD, SYMTLU, PUTX, ERROR
RETURN	XRETN	GETWD, CSORN, ERROR, PUTX
IF	XIF	PUTX, ERROR
ASSIGN	XASGN	GETWD, LITCON, ERROR, SYMTLU, PUTX
BLOCK DATA	XBLOK	PUTX, ERROR
FORMAT	XFMT	CSORN, PUTX
CONTINUE	XCONT	ERROR, PUTX
GO TO	XGO	GETWD, ERROR, SYMTLU, LITCON, PUTX
DATA	XDATA	GETWD, CSORN, ERROR, PUTX
STOP	XSTOP	PUTX
PAUSE	XPUSE	GETWD, ERROR, CSORN, PUTX
END	XEND	ERROR, PUTX

[1]The subroutines used by subroutine XARITH employ the following utility subroutines: GETWD, CSORN, PUTX, COMPAT, ERROR, and SYMTLU.

Table 8. Phase 10 Subroutine Directory

Subroutine	Type	Function
CDOPAR	Utility (entry placement)	Constructs information table entries and pushdown table entries for the index initial value, index increment, and index maximum value appearing in DO statements.
COMAST	Arithmetic	Develops intermediate text and builds information table entries for variables and constants connected by a comma or an asterisk delimiter.
COMPAT	Utility (collection)	Places variable names on word buundaries for comparison to other variable names.
CLOSE	Utility (text generation)	Generates the text entry that signifies the end of the intermediate text representation of a source statement.
CSORN	Utility (entry placement)	Directs the entering of variables and constants into the information table.
DSPTCH	Dispatcher	Control 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.
ERROR	Utility (entry placement)	Builds error table entries for the syntactical errors detected by phase 10 and places them into the error table.
GENDO	Utility (text generation)	Generates the intermediate text required to increment a DO index and to test the index against its maximum.
GETCD	Preparatory	Reads, lists (if requested), packs, and classifies each source statement.
GETWD	Utility (collection)	Obtains the next group of characters in the source statement being processed.
GRPKEQ	Arithmetic	Develops intermediate text and builds information table entries for variables and constants connected by an equal sign or a group mark (end of statement symbol).
INTCON	Utility (conversion)	Calls subroutine LITCON to convert a constant and then verifies that the converted constant is of integer mode.
LABTLU	Utility (entry placement)	Places statement number entries into the information table.
LITCON	Utility (conversion)	Converts integer, real, and complex constants to their binary equivalents.
MINSLS	Arithmetic	Develops intermediate text and builds information table entries for variables and constants connected by a minus or slash delimiter.
PERLOG	Arithmetic	Develops intermediate text and builds information table entries for variables

		and constants connected by a period delimiter.
PH10	Utility (common data area)	Phase 10 COMMON area.
PH10A	Utility (common data area)	Phase 10 COMMON area.
PUTX	Utility (entry placement)	Places text entries into the appropriate sub-blocks, obtains the next operator of the source statement, and places the operator into the text entry work area.
RTPRQT	Arithmetic	Develops intermediate text and builds information table entries for variables and constants connected by a right parenthesis or a quote delimiter.
SYMTLU	Utility (entry placement)	Places the dictionary entries constructed for the variables and constants of the source module into the information table.
TXTBLD	Arithmetic	Develops intermediate text and builds information table entries for variables and constants connected by a left parenthesis, or for complex constants.
XARITH	Arithmetic	Controls the processing of arithmetic statements, CALL arguments, expressions appearing in IF statements, I/O list items, simple variable and array names appearing in NAMELIST statements, complex literals appearing in DATA statements, and arithmetic expressions appearing in statement functions. Subroutine XARITH scans the expression and passes control to one of the following supporting subroutines, depending on the nature of the delimiter recognized: COMAST, GRPKEQ, MINSLS, PERLOG, RTPRQT, and TXTBLD.
XASF	Arithmetic	Scans the portion of a statement function to the left of the equal sign, obtains each dummy argument, and assigns it a sequence number.
XASF2	Arithmetic	Insures that all dummy arguments appearing in the argument list of a statement function are used in the expression to the right of the equal sign in that statement function.
XASGN	Key Word (table entry and text)	Develops an intermediate text representation of the ASSIGN statement, constructs information table entries for its operands, and analyzes the ASSIGN statement for syntactical errors.
XBCKRW	Key Word (table entry and text)	Develops intermediate text representations of the BACKSPACE, REWIND, and END FILE statements, builds information table entries for the operands of these statements, and analyzes these statements for syntactical errors.
XBLOK	Key Word (table entry and text)	Develops an intermediate text representation of the BLOCK DATA statement, set a switch in the communication table to indicate that a BLOCK DATA subprogram is being

		compiled, and analyzes the BLOCK DATA statement for syntactical errors.
XCLASS	Utility (text generation)	Generates intermediate text for statement numbers.
XCOMON	Key Word (table entry)	Constructs information table entries for block names, variables, and arrays appearing in COMMON statements, chains common block name entries and associated variables and arrays together, and analyzes COMMON statements for syntactical errors.
XCONT	Key Word (table entry and text)	Develops and intermediate text representation of the CONTINUE statement, and verifies that there is a statement number associated with it.
XDATA	Key Word (table entry and text)	Develops an intermediate text representation of the DATA statement, constructs information table entries for the operands of the DATA statement, processes the data specifications in TYPE statements, and analyzes DATA statements for syntactical errors.
XDIM	Key Word (table entry)	Constructs information table entries for the arrays appearing in DIMENSION, COMMON; and TYPE statements, and analyzes arrays for syntactical errors.
XDO	Key Word (table entry and text)	Develops, with the aid of subroutines CDOPAR and GENDO, the intermediate text required to control a DO loop.
XEND	Key Word (table entry and text)	Develops an intermediate text representation of the END statement and analyzes the END statement for syntactical errors.
XEQUI	Key Word (table entry)	Builds information table entries for equivalence groups and their associated variables, chains equivalence groups and associated variables together, and analyzes EQUIVALENCE statements for syntactical errors.
XEXT	Key Word (table entry)	Constructs information table entries for the subprogram names appearing in the EXTERNAL statement, signals the subprograms as external, and analyzes the EXTERNAL statement for syntactical errors.
XFMT	Key Word (table entry and text)	Develops an intermediate text representation of the FORMAT statement.
XGO	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.
XIF	Key Word (table entry and text)	Develops an intermediate text representation of that portion of IF statements which precedes the opening parenthesis and passes control to subroutine XARITH to complete the processing of these statements.

XIMPC	Key Word (special)	Sets the type of the variables beginning with the characters stated in the IMPLICIT statement according to the type specifications stated in the IMPLICIT statement, and analyzes the IMPLICIT statement for syntactical errors.
XIMPD	Utiltiy (text generation)	Develops intermediate text representations of implied DO's appearing in I/O statements.
XIOOP	Key Word (table entry and text)	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.)
XNMLST	Key Word (table entry and text)	Develops an intermediate text representation of the NAMELIST statement and constructs information table entries for its operands. (Passes control to subroutine XARITH to process the simple variable of array names.)
XPUSE	Key Word (table entry and text)	Develops an intermediate text representation of the PAUSE statement, constructs information table entries for its operands (if any), and analyzes the PAUSE statement for syntactical errors.
XRETN	Key Word (table entry and text	Develops an intermediate text representation of the RETURN statement, constructs information table entries for its operands (if any), and analyzes the RETURN statement for syntactical errors.
XSTOP	Key Word (table entry and text)	Develops an intermediate text representation of the STOP statement and analyzes that statement for syntactical errors.
XSTRUC		Dummy key word subroutine.
XSUBPG	Key Word (table entry and text)	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 to process the arguments appearing in CALL statements.)
XTYPE	Key Word (table entry and text)	Develops intermediate text representations of TYPE statements, constructs information table entries for their operands, and analyzes the TYPE statements for syntactical errors.

Chart 04. Phase 15 Overall Logic

```
  ****A3**********
 *                *         SEE TABLE 9 FOR A
 *    FROM FSD    *         BRIEF DESCRIPTION
 *                *         OF THE SUBROUTINES
  ****************          OF PHASE 15
         |
         |
         |
         |
         |
         V
 *****B3**********
 *STALL      05B3*
 *-*-*-*-*-*-*-*-*
 *    PROCESS    *
 *  COMMON AND   *
 *  EQUIVALENCE  *
 *****************
         |
         |
         |
         |
         V
 *****C3**********
 *PHAZ15     06B2*
 *-*-*-*-*-*-*-*-*
 *    PROCESS    *
 *   PHASE 10    *
 *     TEXT      *
 *****************
         |
         |
         |
         |
         V
 *****D3**********
 *CORAL      09B2*
 *-*-*-*-*-*-*-*-*
 *   RELATIVE    *
 *    ADDRESS    *
 *  ASSIGNMENT   *
 *****************
         |
         |
         |
         |
         V
  ****E3*********
 *   TO PHASE    *
 *  20 VIA FSD   *
 *               *
  ***************
```

Chart 05. STALL Overall Logic

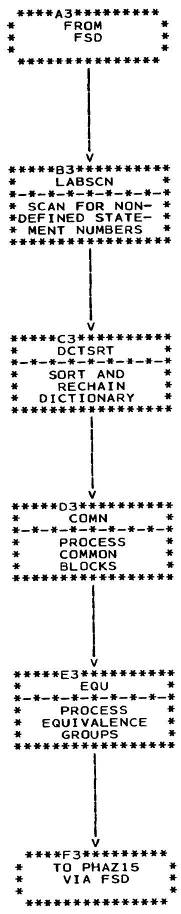
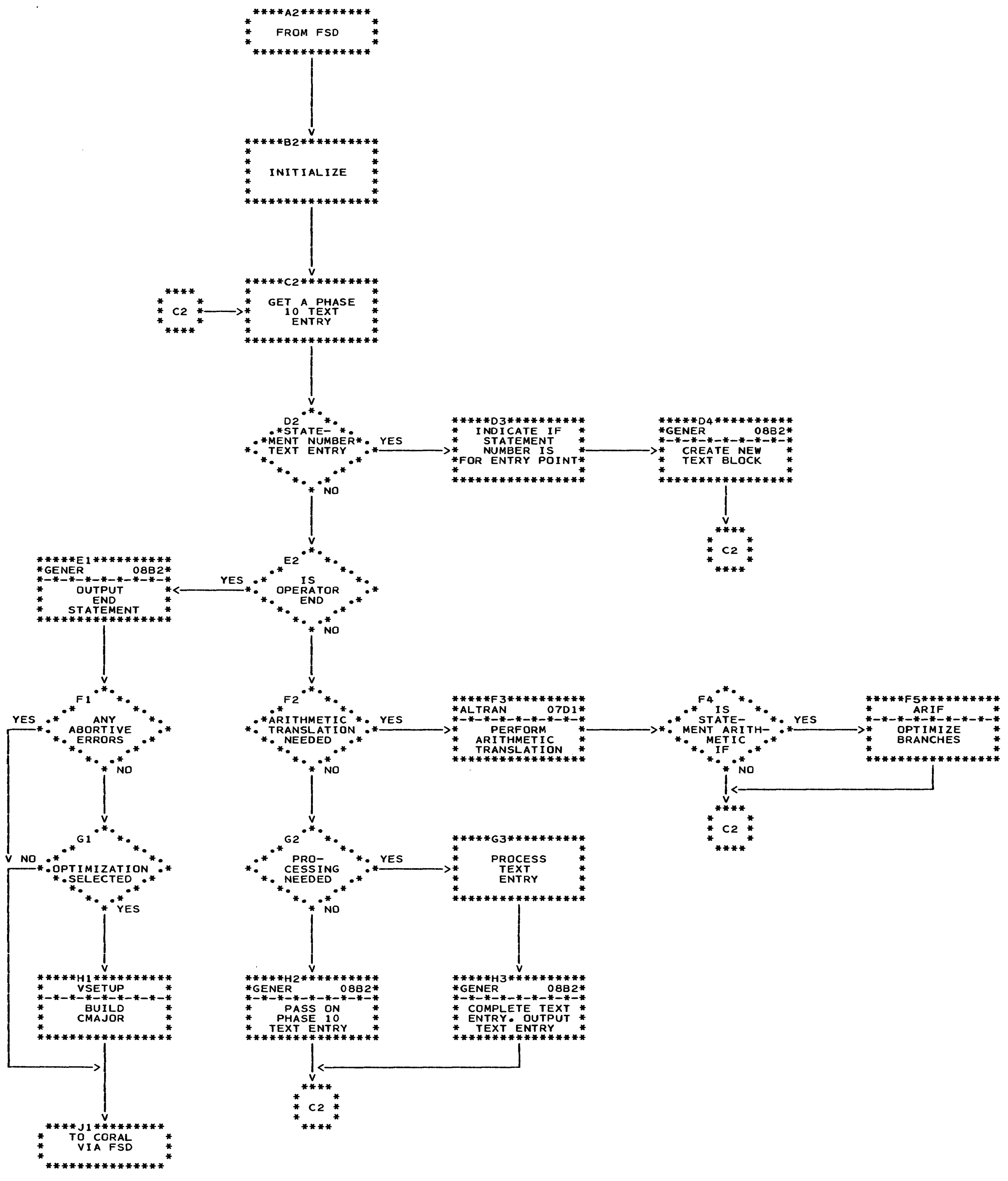
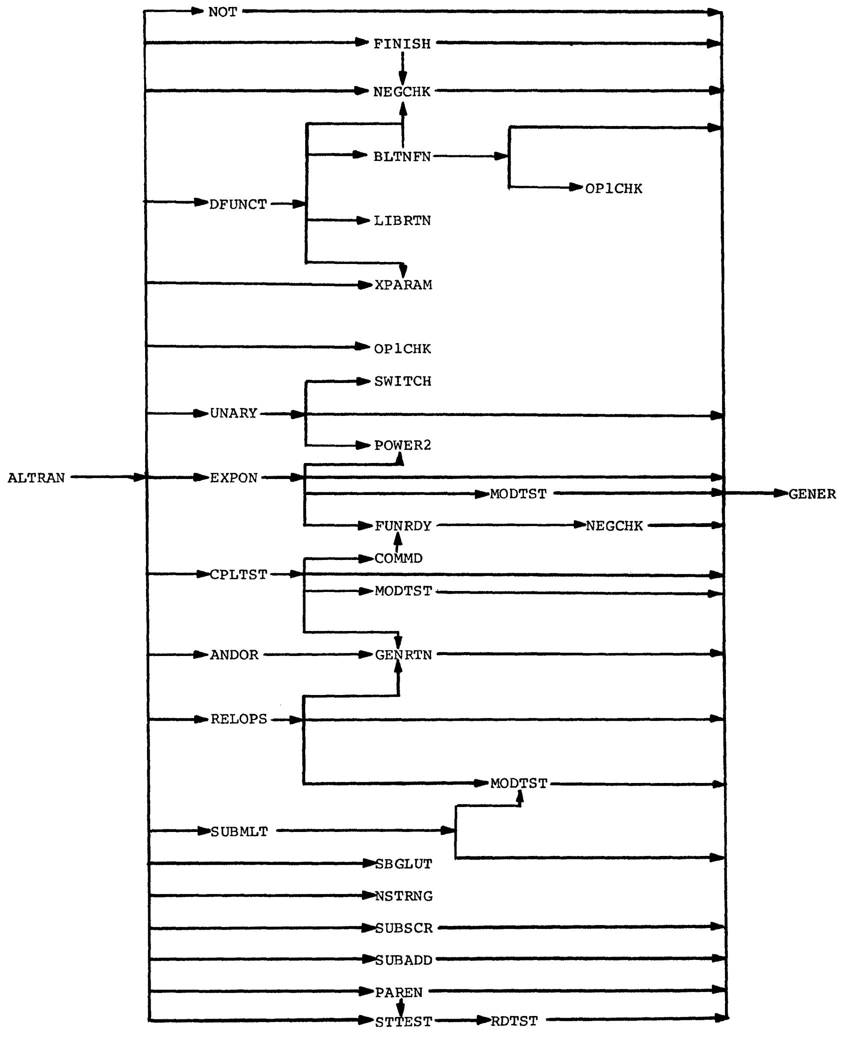

A3
FROM FSD
B3
LABSCN
SCAN FOR NON-DEFINED STATE-MENT NUMBERS
C3
DCTSRT
SORT AND RECHAIN DICTIONARY
D3
COMN
PROCESS COMMON BLOCKS
E3
EQU
PROCESS EQUIVALENCE GROUPS
F3
TO PHAZ15 VIA FSD

Chart 06. PHAZ15 Overall Logic



A2 FROM FSD
B2 INITIALIZE
C2
C2 GET A PHASE 10 TEXT ENTRY
D2 STATEMENT NUMBER TEXT ENTRY
YES
NO
D3 INDICATE IF STATEMENT NUMBER IS FOR ENTRY POINT
D4 GENER 08B2 CREATE NEW TEXT BLOCK
C2
E2 IS OPERATOR END
YES
NO
E1 GENER 08B2 OUTPUT END STATEMENT
F1 ANY ABORTIVE ERRORS
YES
NO
F2 ARITHMETIC TRANSLATION NEEDED
YES
NO
F3 ALTRAN 07D1 PERFORM ARITHMETIC TRANSLATION
F4 IS STATEMENT ARITHMETIC IF
YES
NO
F5 ARIF OPTIMIZE BRANCHES
C2
G1 OPTIMIZATION SELECTED
NO
YES
G2 PROCESSING NEEDED
YES
NO
G3 PROCESS TEXT ENTRY
H1 VSETUP BUILD CMAJOR
H2 GENER 08B2 PASS ON PHASE 10 TEXT ENTRY
H3 GENER 08B2 COMPLETE TEXT ENTRY. OUTPUT TEXT ENTRY
C2
J1 TO CORAL VIA FSD

Chart 07. ALTRAN Control Flow

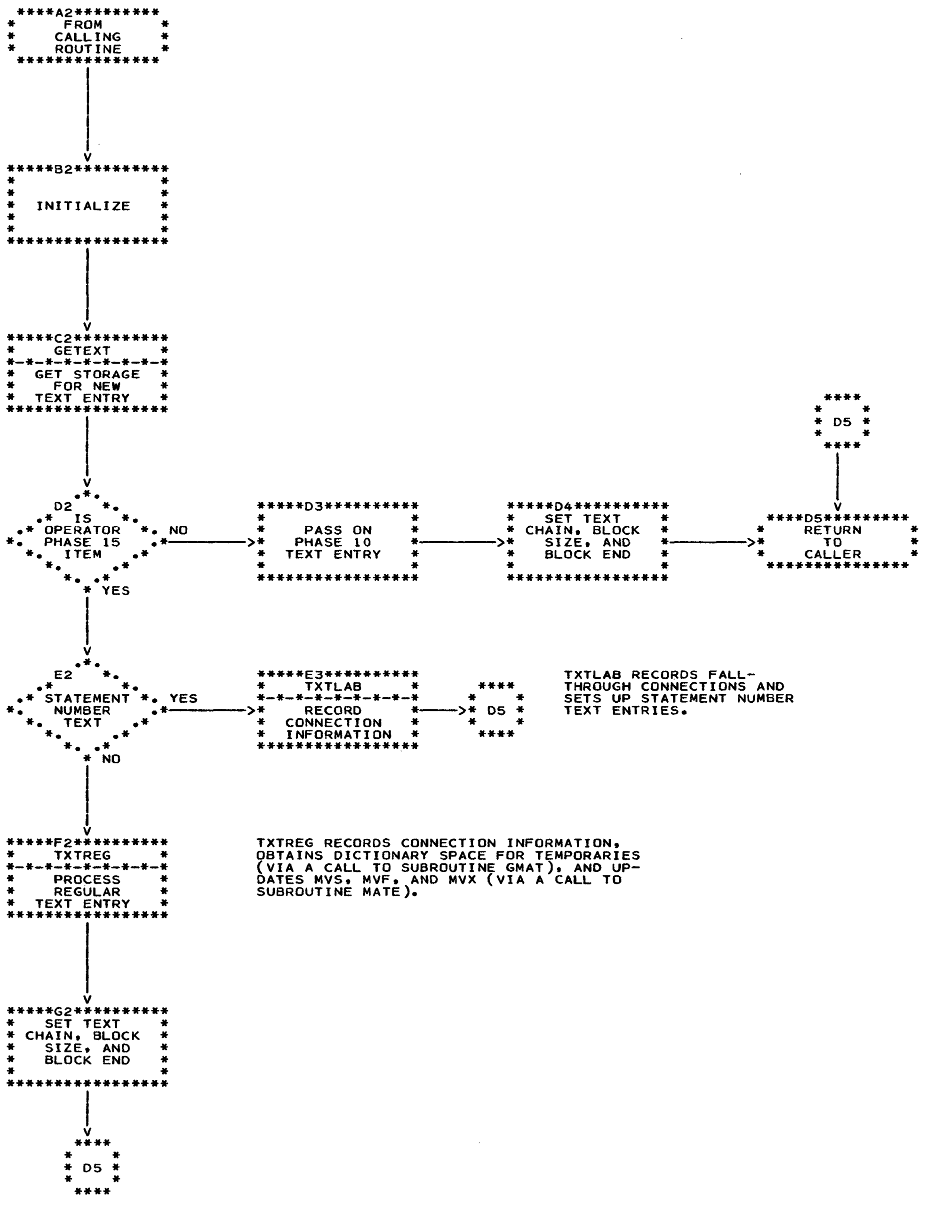


NOTE: The logic and flow of the arithmetic translator is too complex to be represented on one or two conventional flowcharts. Chart 07 indicates the relationship between the arithmetic translator (subroutine ALTRAN) and its lower-level subroutines. An arrow flowing between two subroutines indicates that 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 - Text Generation


A2
FROM CALLING ROUTINE
B2
INITIALIZE
C2
GETEXT
GET STORAGE FOR NEW TEXT ENTRY
D2
IS OPERATOR PHASE 15 ITEM
NO
YES
D3
PASS ON PHASE 10 TEXT ENTRY
D4
SET TEXT CHAIN, BLOCK SIZE, AND BLOCK END
D5
D5
RETURN TO CALLER
E2
STATEMENT NUMBER TEXT
YES
NO
E3
TXTLAB
RECORD CONNECTION INFORMATION
D5
F2
TXTREG
PROCESS REGULAR TEXT ENTRY
G2
SET TEXT CHAIN, BLOCK SIZE, AND BLOCK END
D5

TXTLAB RECORDS FALL-THROUGH CONNECTIONS AND SETS UP STATEMENT NUMBER TEXT ENTRIES.

TXTREG RECORDS CONNECTION INFORMATION, OBTAINS DICTIONARY SPACE FOR TEMPORARIES (VIA A CALL TO SUBROUTINE GMAT), AND UPDATES MVS, MVF, AND MVX (VIA A CALL TO SUBROUTINE MATE).

Chart 09. CORAL Overall Logic

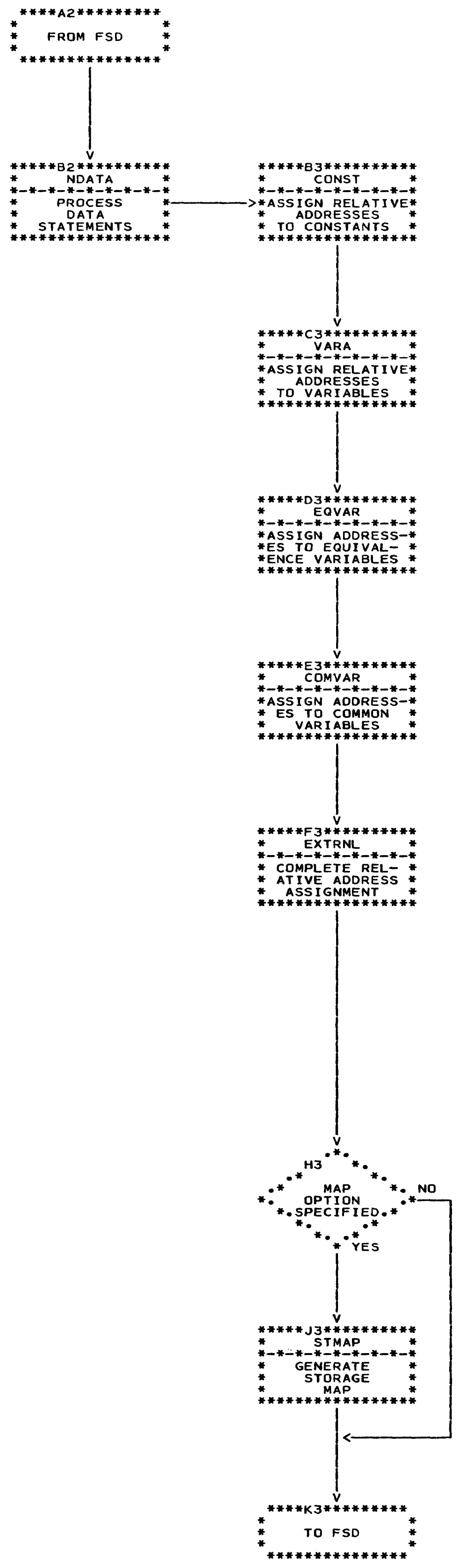

A2
FROM FSD
B2
NDATA
PROCESS DATA STATEMENTS
B3
CONST
ASSIGN RELATIVE ADDRESSES TO CONSTANTS
C3
VARA
ASSIGN RELATIVE ADDRESSES TO VARIABLES
D3
EQVAR
ASSIGN ADDRESS-ES TO EQUIVAL-ENCE VARIABLES
E3
COMVAR
ASSIGN ADDRESS-ES TO COMMON VARIABLES
F3
EXTRNL
COMPLETE REL-ATIVE ADDRESS ASSIGNMENT
H3
MAP OPTION SPECIFIED
NO
YES
J3
STMAP
GENERATE STORAGE MAP
K3
TO FSD

Table 9. Phase 15 Subroutine Directory

Subroutine	Associated Phase 15 Segment	Function
ADSCAN	CORAL	Scans the adcon table for an address constant that references the relative address computed for a variable.
ALTRAN[1]	PHAZ15	Controls the arithmetic translation process.
ANDOR[1]	PHAZ15	Checks the mode of the arguments passed to it, decomposes IF statements, and generates text entries for AND and OR operations.
ARIF	PHAZ15	Optimizes the coding derived from the branching portion of an arithmetic IF statement.
BLTNFN[1]	PHAZ15	Determines whether or not a given name represents a valid in-line function, and generates phase 15 text for the referenced in-line function.
BSIZE	STALL	Computes the size (in bytes) of a variable or array based on its mode and dimensions (if any).
C1520		Common data area used by phases 15 and 20.
CMSIZE	CORAL	Checks the displacement computed by subroutine SPAN to see if it lies within the range of 0 to 4096 bytes.
COMMD[1]	PHAZ15	Generates the text required for complex multiplication or division (i.e., a call to a library routine).
COMN	STALL	Processes the common table entries constructed by phase 10 for the operands appearing in COMMON statements.
COMVAR	CORAL	Assigns relative addresses to common variables and variables equivalenced into common.
CONST	CORAL	Assigns relative addresses to all constants in the dictionary.
CORAL	CORAL	Controls the relative address assignment function of phase 15.
CPLTST[1]	PHAZ15	Checks triplets for complex operands and controls text generation for the same.
DATACH	CORAL	Chains the data text created by subroutine NDATA in the order in which it will be processed by phase 25.
DCTSRT	STALL	Sorts the dictionary constructed by phase 10.
DFUNCT[1]	PHAZ15	Determines if a reference is to an in-line, library, or external function.
DUMP15	PHAZ15	Records errors detected during PHAZ15 processing.
EQU	STALL	Establishes a "head" for each equivalence group and computed the displacement of each variable in the group from the group head.
EQVAR	CORAL	Assigns relative addresses to equivalence variables except those that are equivalenced into common.
ERDATA	CORAL	Places entries into the error table for errors detected during the processing of common blocks and equivalence groups.
EXPON[1]	PHAZ15	Generates the text required for exponentiation operations.

EXTRNL	CORAL	Completes the relative address assignment process by reserving address constants for quantities not previously assigned addresses.
FINISH[1]	PHAZ15	Completes the processing required for a statement when its primary adjective code is forced from the pushdown table.
FUNRDY[1]	PHAZ15	Creates pushdown entries for references to implicit library functions.
GENER	PHAZ15	Outputs phase 15 text consisting of unchanged phase 10 text, phase 15 standard text, and phase 15 statement number text.
GENRTN[1]	PHAZ15	Builds appropriate phase 15 text entries for items forced from the pushdown table.
GETEXT	PHAZ15	Provides subroutine GENER with the main storage needed for a text entry.
GMAT	PHAZ15	Creates an abbreviated one-word dictionary entry for temporaries.
IFUNTB		Common data area, which is the FORTRAN supplied subprogram table.
LABSCN	STALL	Scans the statement number entry chain for statement numbers that are referenced, but not defined.
LIBRTN[1]	PHAZ15	Determines if the use of a library routine name is valid, and performs automatic typing where necessary.
LOOKER[1]	PHAZ15	Looks up names in the IFUNTB (subprogram) table.
MATE	PHAZ15	Records usage information in the MVS, MVF, and MVX fields if the optimized path through phase 20 is selected.
MODIFY[1]	PHAZ15	Changes modes for logical expressions.
MODTST[1]	PHAZ15	Checks for mixed-mode conditions in the triplet supplied to it.
NDATA	CORAL	Converts phase 10 data text to phase 15 data text.
NEGCHK[1]	PHAZ15	Checks for negative operands in the argument list of a function.
NSTRNG[1]	PHAZ15	Determines the forcing strength of operators.
OP1CHK[1]	PHAZ15	Determine if operand 1 is to be an actual operand or a temporary.
PAREN[1]	PHAZ15	Removes the (or -(from the pushdown table when the corresponding) is encountered.
PH15		Common data area used by phase 15.
PHAZ15	PHAZ15	Controlling subroutine of PHAZ15 processing.
PHSTAL		Common data area used during relative address assignment.
POWER2[1]	PHAZ15	Determines whether or not the argument passed to it is an integral power of two.
PRTEXT	CORAL	Prints out phase 15 data text.
RDTST[1]	PHAZ15	Builds text for replacement statements (e.g., A=B, A=B(I), A(I)=B, A(I)=B(I)).
RELOPS[1]	PHAZ15	Calls subroutine GENER to output text entries for relational

		operators. (Output may be either a relational or branch operation.)
SBEROR	STALL	Places entries into the error table for errors detected during the processing of COMMON and EQUIVALENCE declarations.
SBGLUT[1]	PHAZ15	Optimizes subscript computations by evaluating subscript constants.
SIZE	CORAL	Computes the total size (in bytes) of a variable or constant.
SPAN	CORAL	Computes the span of an array.
STALL	STALL	Controlling subroutine of STALL processing.
STMAP2	CORAL	Writes a storage map if the MAP option is specified.
STTEST[1]	PHAZ15	Calls RDTST to process replacement statements.
SUBADD[1]	PHAZ15	Generates the text to add the terms in a subscript computation.
SUBMLT[1]	PHAZ15	Generates the text to multiply the first term in a subscript computation by its associated length factor, or, in the case of variable dimension, to multiply the nth dimension by length.
SUBSCR[1]	PHAZ15	Determines if a subscript text entry in the pushdown table should be entered into phase 15 text, and calls subroutine GENER to output the text entry when appropriate.
SWITCH[1]	PHAZ15	Inverts the order of the operands supplied to it.
TESTBN	STALL	Tests the mode and displacement of a variable to determine whether or not a boundary violation exists.
TESTWD	CORAL	Determines whether or not a given variable is to be processed by subroutine VARA.
TXTLAB	PHAZ15	Processes statement number text entries for subroutine GENER; creates entries in RMAJOR.
TXTREG	PHAZ15	Processes standard phase 15 text entries for subroutine GENER and makes RMAJOR entries.
UNARY[1]	PHAZ15	Checks for negativeness in the triplet supplied to it, and modifies the triplet (if negativeness is present) to optimize subsequent code generation. Also detects multiplication and division operations and attempts to implement them by generating shift operations.
VARA	CORAL	Assigns relative addresses to all variables in the dictionary except for variables in COMMON and/or EQUIVALENCE statements, external functions, namelist names, and variables called by name and not by value.
XPARAM[1]	PHAZ15	Inserts the appropriate function operator into phase 15 text and builds the parameter list for the referenced subprogram in the adcon table and in text.

[1]This subroutine is used during arithmetic translation.

Chart 10. Phase 20 Overall Logic

SEE TABLE 11 FOR A BRIEF DESCRIPTION OF THE MAJOR SUBROUTINES OF PHASE 20.

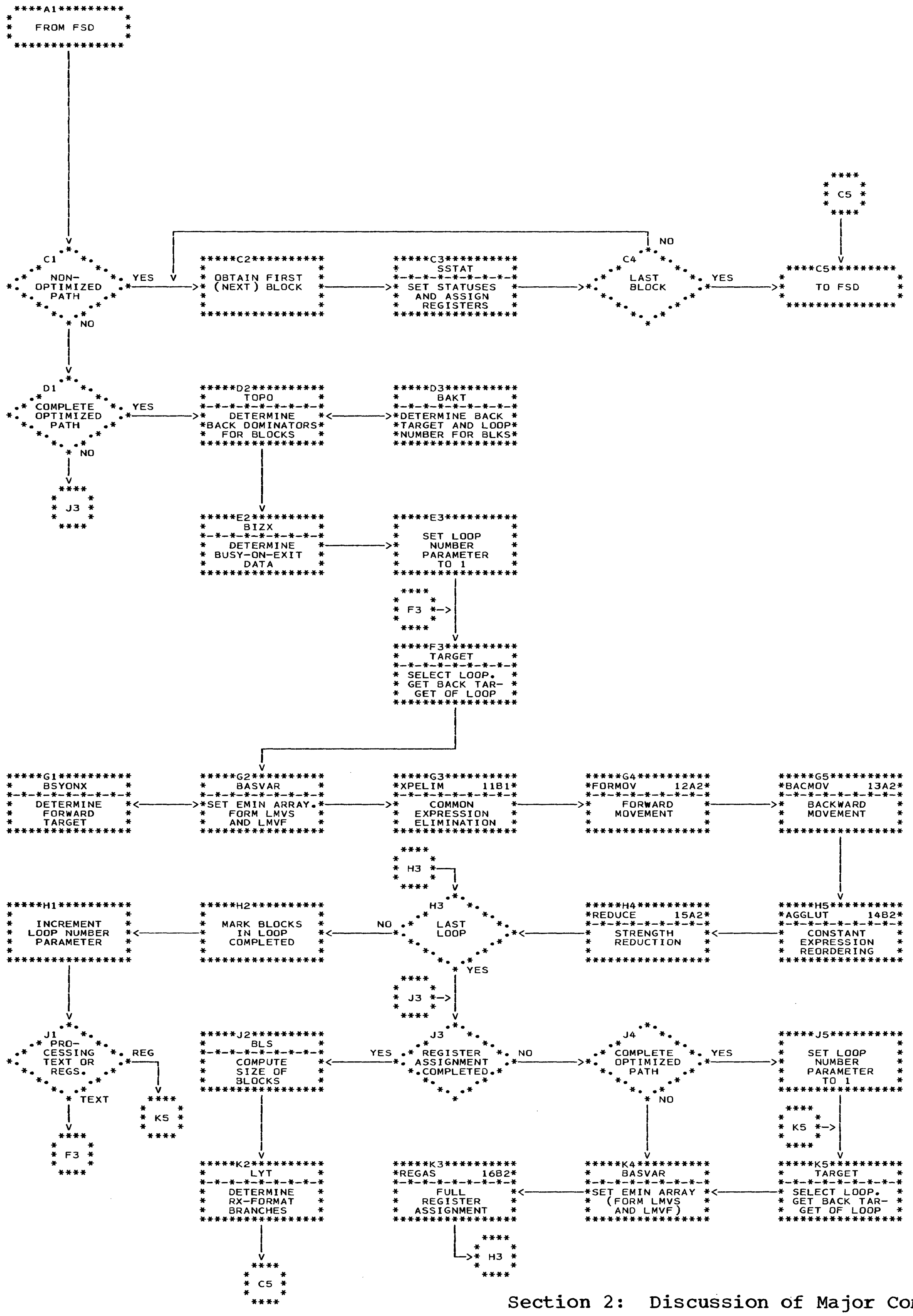

A1 FROM FSD
C1 NON-OPTIMIZED PATH
YES
NO
C2 OBTAIN FIRST (NEXT) BLOCK
C3 SSTAT SET STATUSES AND ASSIGN REGISTERS
C4 LAST BLOCK
C5 TO FSD
C5
D1 COMPLETE OPTIMIZED PATH
J3
D2 TOPO DETERMINE BACK DOMINATORS FOR BLOCKS
D3 BAKT DETERMINE BACK TARGET AND LOOP NUMBER FOR BLKS
E2 BIZX DETERMINE BUSY-ON-EXIT DATA
E3 SET LOOP NUMBER PARAMETER TO 1
F3
F3 TARGET SELECT LOOP. GET BACK TARGET OF LOOP
G1 BSYONX DETERMINE FORWARD TARGET
G2 BASVAR SET EMIN ARRAY. FORM LMVS AND LMVF
G3 XPELIM 11B1 COMMON EXPRESSION ELIMINATION
G4 FORMOV 12A2 FORWARD MOVEMENT
G5 BACMOV 13A2 BACKWARD MOVEMENT
H3
H1 INCREMENT LOOP NUMBER PARAMETER
H2 MARK BLOCKS IN LOOP COMPLETED
H3 LAST LOOP
H4 REDUCE 15A2 STRENGTH REDUCTION
H5 AGGLUT 14B2 CONSTANT EXPRESSION REORDERING
J3
J1 PROCESSING TEXT OR REGS.
REG
TEXT
K5
F3
J2 BLS COMPUTE SIZE OF BLOCKS
J3 REGISTER ASSIGNMENT COMPLETED.
J4 COMPLETE OPTIMIZED PATH
J5 SET LOOP NUMBER PARAMETER TO 1
K5
K2 LYT DETERMINE RX-FORMAT BRANCHES
K3 REGAS 16B2 FULL REGISTER ASSIGNMENT
K4 BASVAR SET EMIN ARRAY (FORM LMVS AND LMVF)
K5 TARGET SELECT LOOP. GET BACK TARGET OF LOOP
H3
C5

Chart 11. Cmmn Xpressn Elmntn (XPELIM)

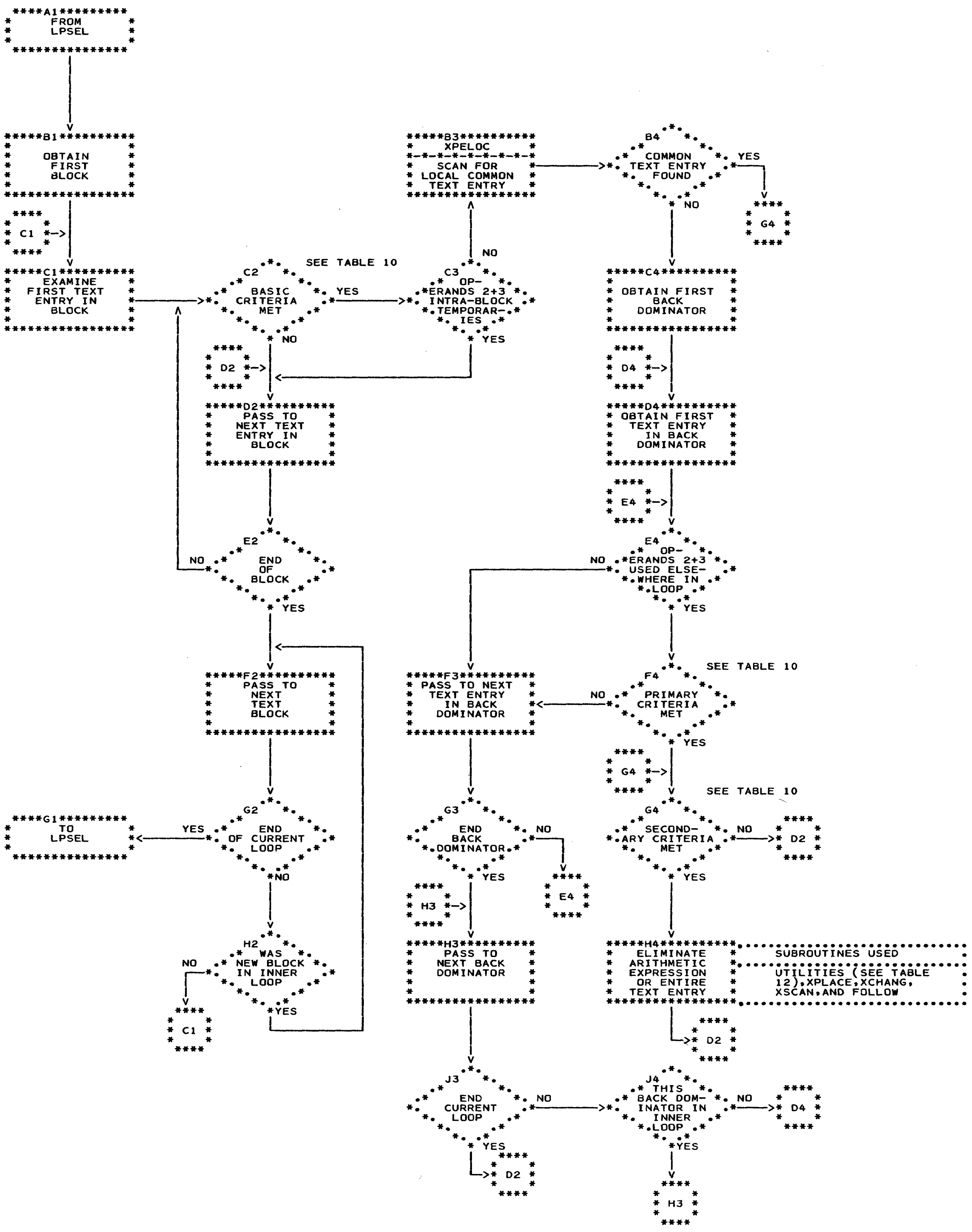

A1 FROM LPSEL
B1 OBTAIN FIRST BLOCK
C1
C1 EXAMINE FIRST TEXT ENTRY IN BLOCK
C2 BASIC CRITERIA MET
SEE TABLE 10
YES
NO
D2
D2 PASS TO NEXT TEXT ENTRY IN BLOCK
E2 END OF BLOCK
NO
YES
F2 PASS TO NEXT TEXT BLOCK
G2 END OF CURRENT LOOP
YES
NO
G1 TO LPSEL
H2 WAS NEW BLOCK IN INNER LOOP
NO
YES
C1
B3 XPELOC SCAN FOR LOCAL COMMON TEXT ENTRY
C3 OP-ERANDS 2+3 INTRA-BLOCK TEMPORAR-IES
NO
YES
F3 PASS TO NEXT TEXT ENTRY IN BACK DOMINATOR
G3 END BACK DOMINATOR
NO
YES
E4
H3
H3 PASS TO NEXT BACK DOMINATOR
J3 END CURRENT LOOP
NO
YES
D2
B4 COMMON TEXT ENTRY FOUND
YES
NO
G4
C4 OBTAIN FIRST BACK DOMINATOR
D4
D4 OBTAIN FIRST TEXT ENTRY IN BACK DOMINATOR
E4
E4 OP-ERANDS 2+3 USED ELSE-WHERE IN LOOP
NO
YES
F4 PRIMARY CRITERIA MET
SEE TABLE 10
NO
YES
G4
G4 SECOND-ARY CRITERIA MET
SEE TABLE 10
NO
D2
YES
H4 ELIMINATE ARITHMETIC EXPRESSION OR ENTIRE TEXT ENTRY
SUBROUTINES USED
UTILITIES (SEE TABLE 12),XPLACE,XCHANG, XSCAN,AND FOLLOW
D2
J4 THIS BACK DOM-INATOR IN INNER LOOP
NO
D4
YES
H3

Chart 12. **Forward Movement** (FORMOV)

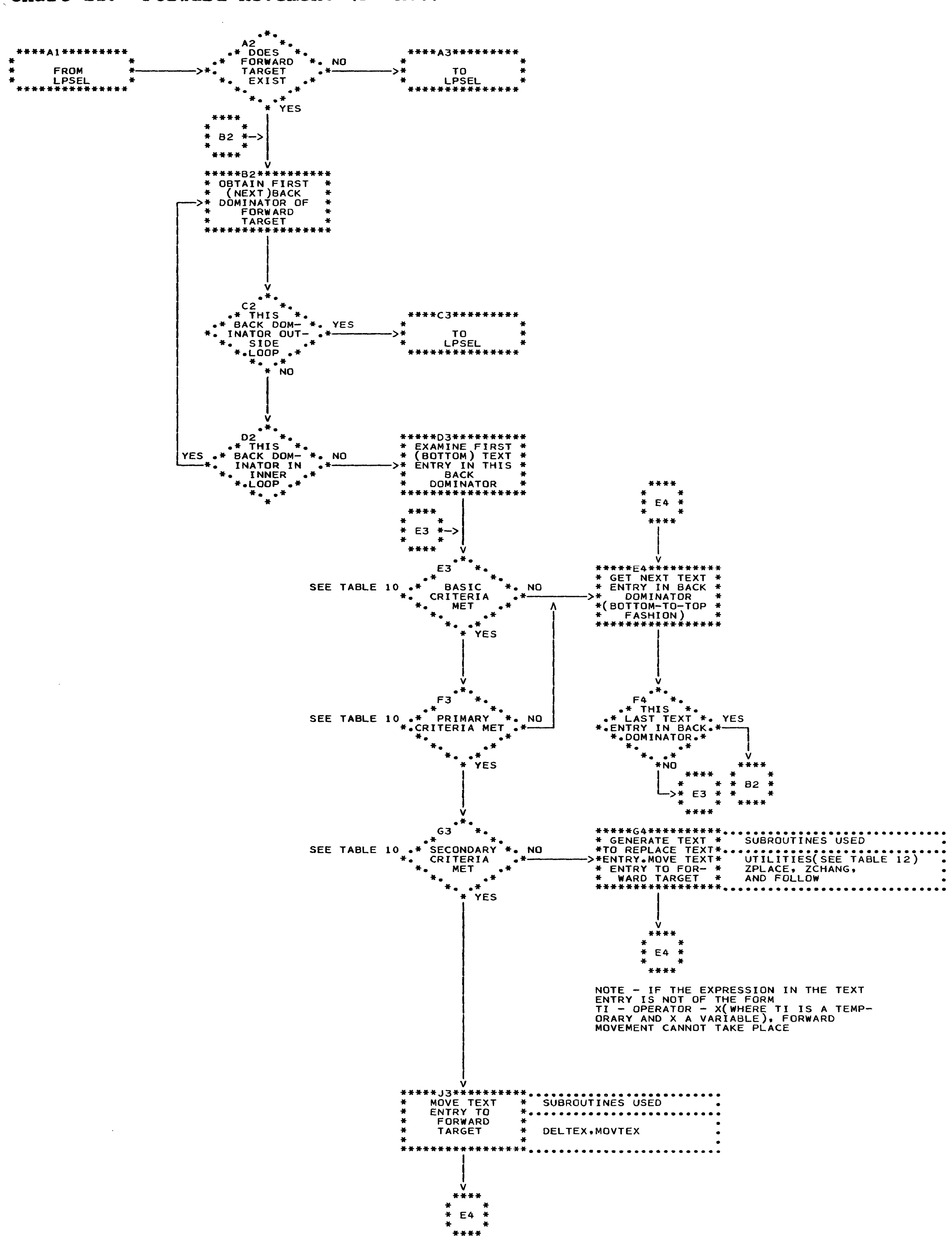
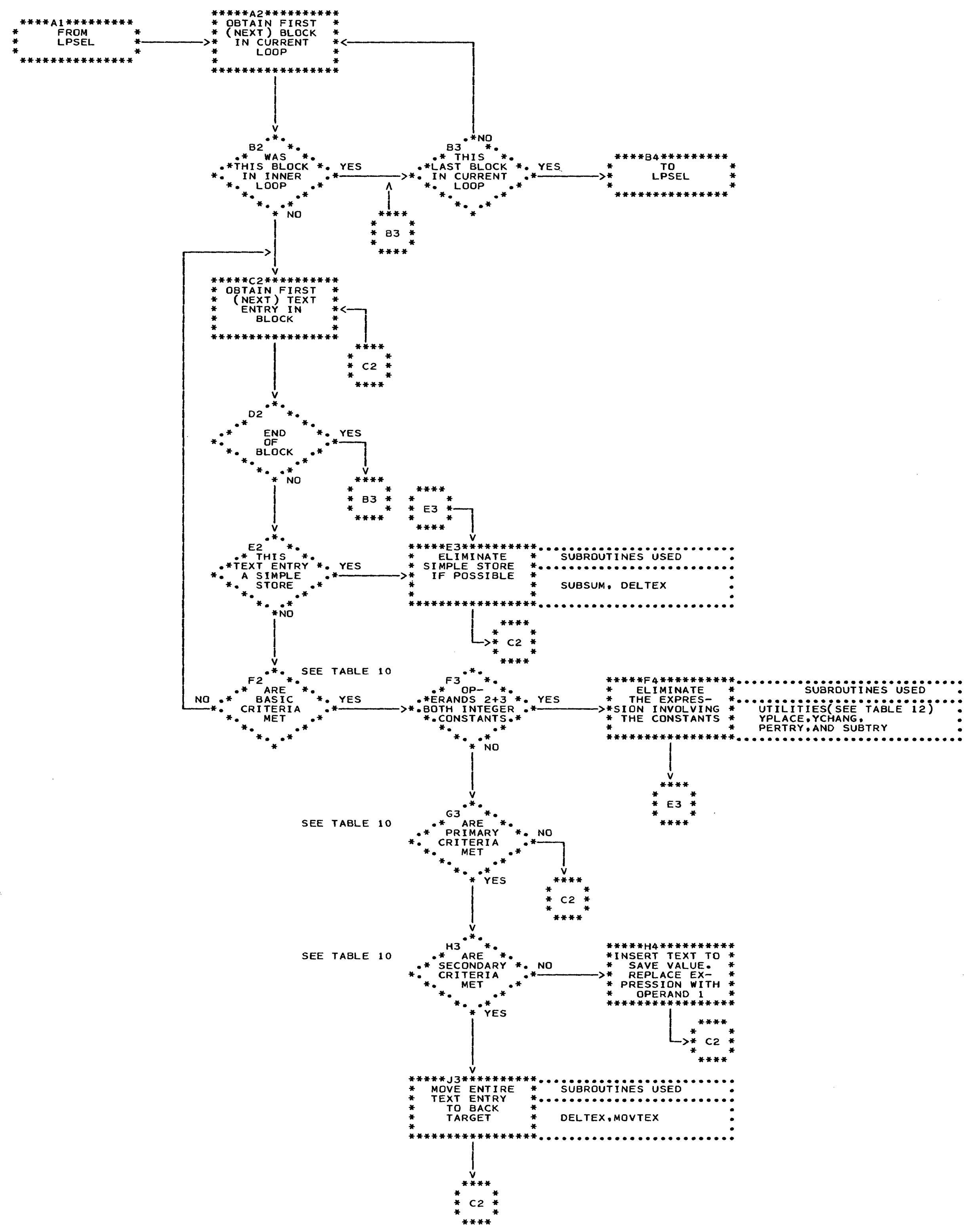

A1 FROM LPSEL
A2 DOES FORWARD TARGET EXIST
NO
A3 TO LPSEL
YES
B2
B2 OBTAIN FIRST (NEXT)BACK DOMINATOR OF FORWARD TARGET
C2 THIS BACK DOMINATOR OUTSIDE LOOP
YES
C3 TO LPSEL
NO
D2 THIS BACK DOMINATOR IN INNER LOOP
YES
NO
D3 EXAMINE FIRST (BOTTOM) TEXT ENTRY IN THIS BACK DOMINATOR
E3
SEE TABLE 10
E3 BASIC CRITERIA MET
NO
YES
E4
E4 GET NEXT TEXT ENTRY IN BACK DOMINATOR (BOTTOM-TO-TOP FASHION)
SEE TABLE 10
F3 PRIMARY CRITERIA MET
NO
YES
F4 THIS LAST TEXT ENTRY IN BACK DOMINATOR
YES
NO
E3
B2
SEE TABLE 10
G3 SECONDARY CRITERIA MET
NO
YES
G4 GENERATE TEXT TO REPLACE TEXT ENTRY. MOVE TEXT ENTRY TO FORWARD TARGET
SUBROUTINES USED
UTILITIES(SEE TABLE 12) ZPLACE, ZCHANG, AND FOLLOW
E4
NOTE - IF THE EXPRESSION IN THE TEXT ENTRY IS NOT OF THE FORM TI - OPERATOR - X(WHERE TI IS A TEMPORARY AND X A VARIABLE), FORWARD MOVEMENT CANNOT TAKE PLACE
J3 MOVE TEXT ENTRY TO FORWARD TARGET
SUBROUTINES USED
DELTEX,MOVTEX
E4

Chart 13. Backward Movement (BACMOV)



A1
FROM LPSEL
A2
OBTAIN FIRST (NEXT) BLOCK IN CURRENT LOOP
B2
WAS THIS BLOCK IN INNER LOOP
YES
NO
B3
THIS LAST BLOCK IN CURRENT LOOP
YES
NO
B3
B4
TO LPSEL
C2
OBTAIN FIRST (NEXT) TEXT ENTRY IN BLOCK
C2
D2
END OF BLOCK
YES
NO
B3
E3
E2
THIS TEXT ENTRY A SIMPLE STORE
YES
NO
E3
ELIMINATE SIMPLE STORE IF POSSIBLE
SUBROUTINES USED
SUBSUM, DELTEX
C2
SEE TABLE 10
F2
ARE BASIC CRITERIA MET
NO
YES
F3
OP- ERANDS 2+3 BOTH INTEGER CONSTANTS
YES
NO
F4
ELIMINATE THE EXPRES- SION INVOLVING THE CONSTANTS
SUBROUTINES USED
UTILITIES(SEE TABLE 12) YPLACE,YCHANG, PERTRY,AND SUBTRY
E3
SEE TABLE 10
G3
ARE PRIMARY CRITERIA MET
NO
YES
C2
SEE TABLE 10
H3
ARE SECONDARY CRITERIA MET
NO
YES
H4
INSERT TEXT TO SAVE VALUE. REPLACE EX- PRESSION WITH OPERAND 1
C2
J3
MOVE ENTIRE TEXT ENTRY TO BACK TARGET
SUBROUTINES USED
DELTEX,MOVTEX
C2

Chart 14. Constant Xprssn Reordrng (AGGLUT)

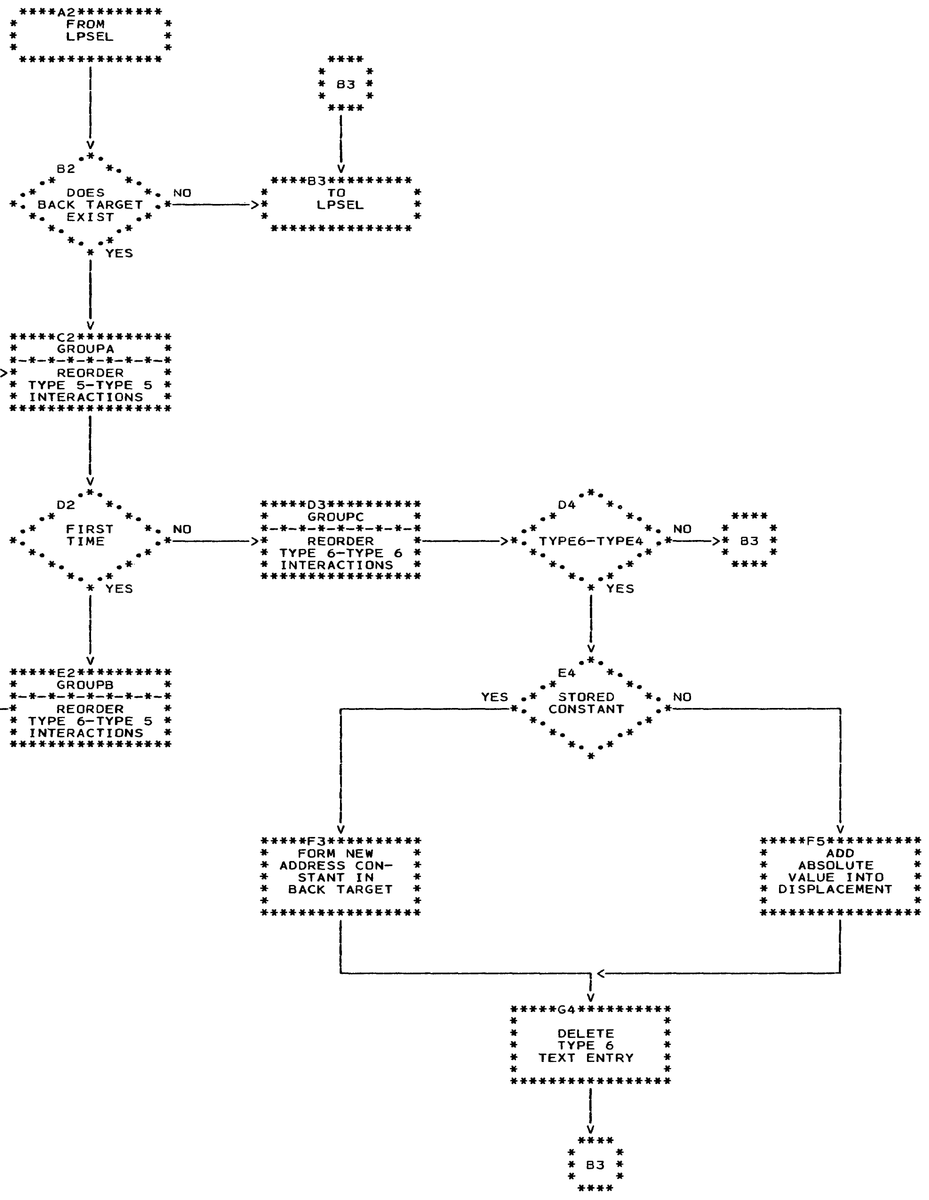

A2 FROM LPSEL
B3
B2 DOES BACK TARGET EXIST
NO
YES
B3 TO LPSEL
C2 GROUPA REORDER TYPE 5-TYPE 5 INTERACTIONS
D2 FIRST TIME
NO
YES
D3 GROUPC REORDER TYPE 6-TYPE 6 INTERACTIONS
D4 TYPE6-TYPE4
NO
B3
YES
E2 GROUPB REORDER TYPE 6-TYPE 5 INTERACTIONS
E4 STORED CONSTANT
YES
NO
F3 FORM NEW ADDRESS CON- STANT IN BACK TARGET
F5 ADD ABSOLUTE VALUE INTO DISPLACEMENT
G4 DELETE TYPE 6 TEXT ENTRY
B3

Chart 15. Strength Reduction (REDUCE)

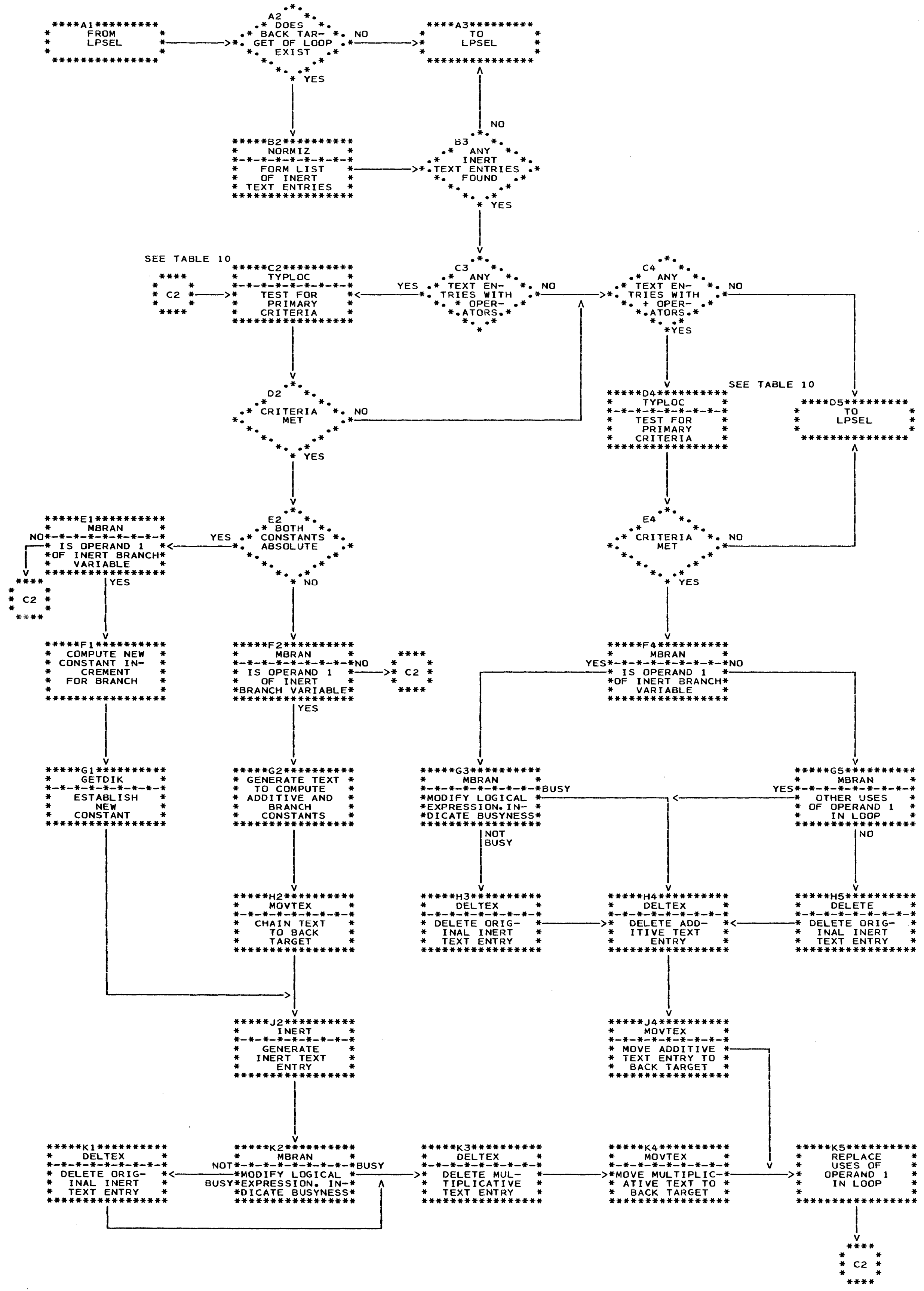
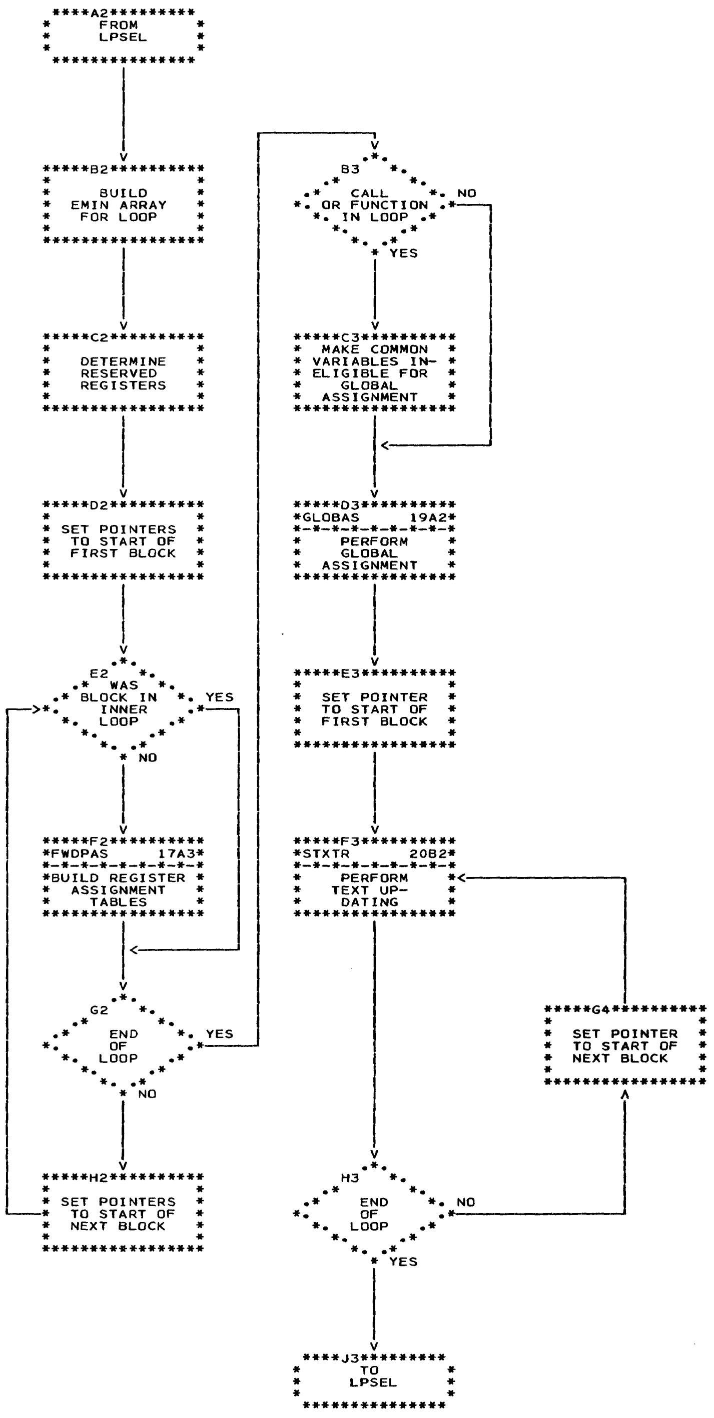
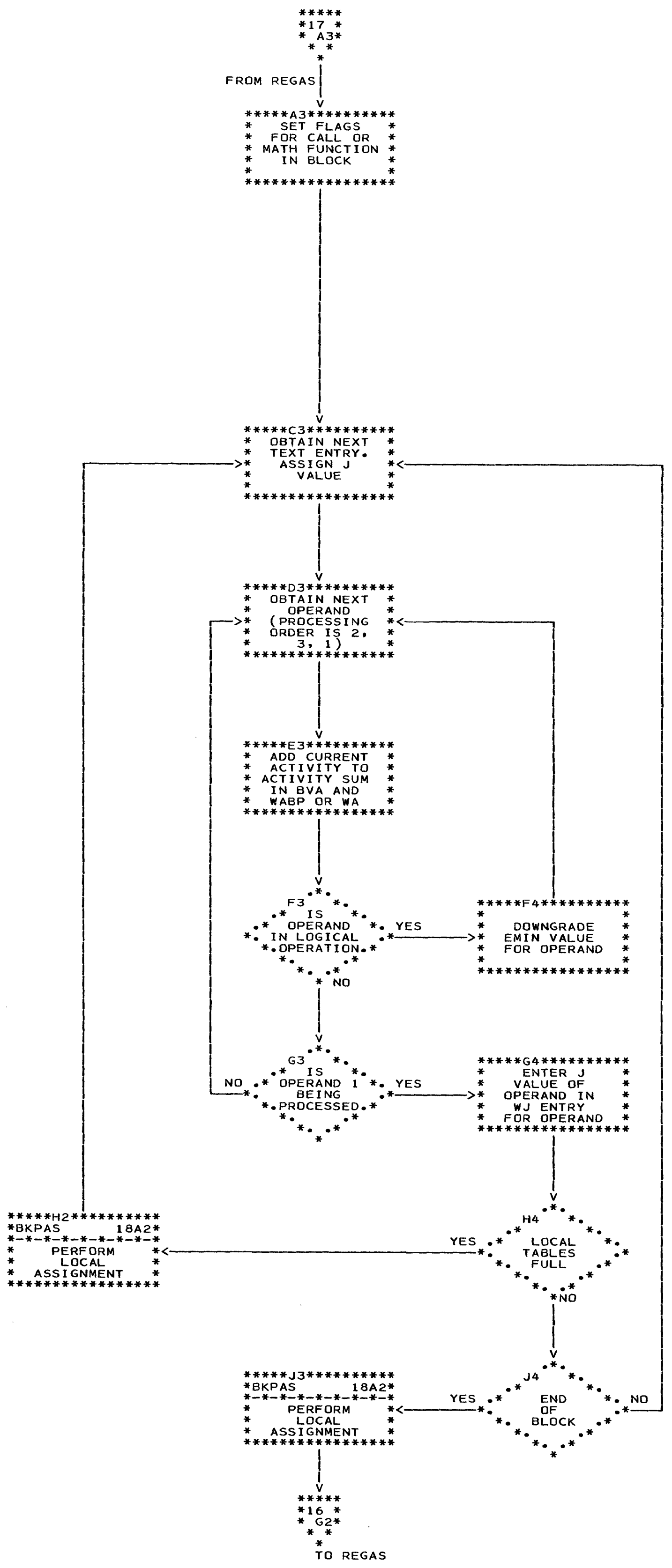

A1 FROM LPSEL
A2 DOES BACK TARGET OF LOOP EXIST
NO
YES
A3 TO LPSEL
B2 NORMIZ FORM LIST OF INERT TEXT ENTRIES
B3 ANY INERT TEXT ENTRIES FOUND
NO
YES
SEE TABLE 10
C2
C2 TYPLOC TEST FOR PRIMARY CRITERIA
C3 ANY TEXT ENTRIES WITH + OPERATORS
YES
NO
C4 ANY TEXT ENTRIES WITH + OPERATORS
NO
YES
D2 CRITERIA MET
NO
YES
SEE TABLE 10
D4 TYPLOC TEST FOR PRIMARY CRITERIA
D5 TO LPSEL
E1 MBRAN IS OPERAND 1 OF INERT BRANCH VARIABLE
NO
YES
C2
E2 BOTH CONSTANTS ABSOLUTE
YES
NO
E4 CRITERIA MET
NO
YES
F1 COMPUTE NEW CONSTANT INCREMENT FOR BRANCH
F2 MBRAN IS OPERAND 1 OF INERT BRANCH VARIABLE
NO
YES
C2
F4 MBRAN IS OPERAND 1 OF INERT BRANCH VARIABLE
YES
NO
G1 GETDIK ESTABLISH NEW CONSTANT
G2 GENERATE TEXT TO COMPUTE ADDITIVE AND BRANCH CONSTANTS
G3 MBRAN MODIFY LOGICAL EXPRESSION. INDICATE BUSYNESS
BUSY
NOT BUSY
G5 MBRAN OTHER USES OF OPERAND 1 IN LOOP
YES
NO
H2 MOVTEX CHAIN TEXT TO BACK TARGET
H3 DELTEX DELETE ORIGINAL INERT TEXT ENTRY
H4 DELTEX DELETE ADDITIVE TEXT ENTRY
H5 DELETE DELETE ORIGINAL INERT TEXT ENTRY
J2 INERT GENERATE INERT TEXT ENTRY
J4 MOVTEX MOVE ADDITIVE TEXT ENTRY TO BACK TARGET
K1 DELTEX DELETE ORIGINAL INERT TEXT ENTRY
K2 MBRAN MODIFY LOGICAL EXPRESSION. INDICATE BUSYNESS
NOT BUSY
BUSY
K3 DELTEX DELETE MULTIPLICATIVE TEXT ENTRY
K4 MOVTEX MOVE MULTIPLICATIVE TEXT TO BACK TARGET
K5 REPLACE USES OF OPERAND 1 IN LOOP
C2

Chart 16. Full Register Assignment (REGAS)



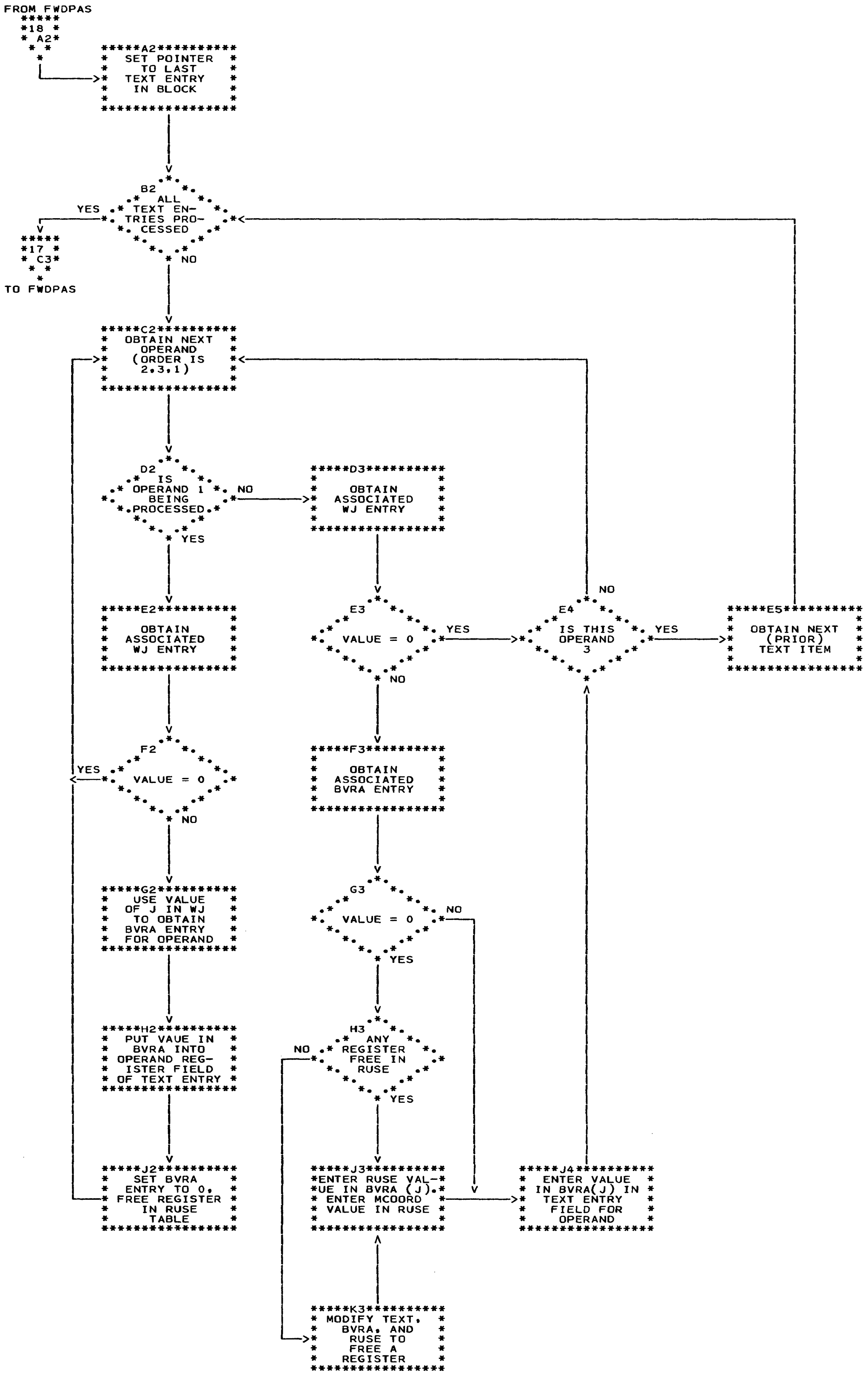
A2 FROM LPSEL
B2 BUILD EMIN ARRAY FOR LOOP
C2 DETERMINE RESERVED REGISTERS
D2 SET POINTERS TO START OF FIRST BLOCK
E2 WAS BLOCK IN INNER LOOP
YES
NO
F2 FWDPAS 17A3 BUILD REGISTER ASSIGNMENT TABLES
G2 END OF LOOP
YES
NO
H2 SET POINTERS TO START OF NEXT BLOCK
B3 CALL OR FUNCTION IN LOOP
NO
YES
C3 MAKE COMMON VARIABLES IN-ELIGIBLE FOR GLOBAL ASSIGNMENT
D3 GLOBAS 19A2 PERFORM GLOBAL ASSIGNMENT
E3 SET POINTER TO START OF FIRST BLOCK
F3 STXTR 20B2 PERFORM TEXT UP-DATING
G4 SET POINTER TO START OF NEXT BLOCK
H3 END OF LOOP
NO
YES
J3 TO LPSEL

Chart 17. Table Building (FWDPAS)



17 A3
FROM REGAS
A3 SET FLAGS FOR CALL OR MATH FUNCTION IN BLOCK
C3 OBTAIN NEXT TEXT ENTRY. ASSIGN J VALUE
D3 OBTAIN NEXT OPERAND (PROCESSING ORDER IS 2, 3, 1)
E3 ADD CURRENT ACTIVITY TO ACTIVITY SUM IN BVA AND WABP OR WA
F3 IS OPERAND IN LOGICAL OPERATION
YES
NO
F4 DOWNGRADE EMIN VALUE FOR OPERAND
G3 IS OPERAND 1 BEING PROCESSED
NO
YES
G4 ENTER J VALUE OF OPERAND IN WJ ENTRY FOR OPERAND
H2 BKPAS 18A2 PERFORM LOCAL ASSIGNMENT
H4 LOCAL TABLES FULL
YES
NO
J4 END OF BLOCK
YES
NO
J3 BKPAS 18A2 PERFORM LOCAL ASSIGNMENT
16 G2
TO REGAS

Chart 18. Local Assignment (BKPAS)



FROM FWDPAS
18 A2
A2 SET POINTER TO LAST TEXT ENTRY IN BLOCK
B2 ALL TEXT ENTRIES PROCESSED
YES
NO
17 C3
TO FWDPAS
C2 OBTAIN NEXT OPERAND (ORDER IS 2,3,1)
D2 IS OPERAND 1 BEING PROCESSED
NO
YES
D3 OBTAIN ASSOCIATED WJ ENTRY
E2 OBTAIN ASSOCIATED WJ ENTRY
E3 VALUE = 0
YES
NO
E4 IS THIS OPERAND 3
YES
NO
E5 OBTAIN NEXT (PRIOR) TEXT ITEM
F2 VALUE = 0
YES
NO
F3 OBTAIN ASSOCIATED BVRA ENTRY
G2 USE VALUE OF J IN WJ TO OBTAIN BVRA ENTRY FOR OPERAND
G3 VALUE = 0
NO
YES
H2 PUT VALUE IN BVRA INTO OPERAND REGISTER FIELD OF TEXT ENTRY
H3 ANY REGISTER FREE IN RUSE
NO
YES
J2 SET BVRA ENTRY TO 0, FREE REGISTER IN RUSE TABLE
J3 ENTER RUSE VALUE IN BVRA (J). ENTER MCOORD VALUE IN RUSE
J4 ENTER VALUE IN BVRA(J) IN TEXT ENTRY FIELD FOR OPERAND
K3 MODIFY TEXT, BVRA, AND RUSE TO FREE A REGISTER

Chart 19. Global Assignment (GLOBAS)

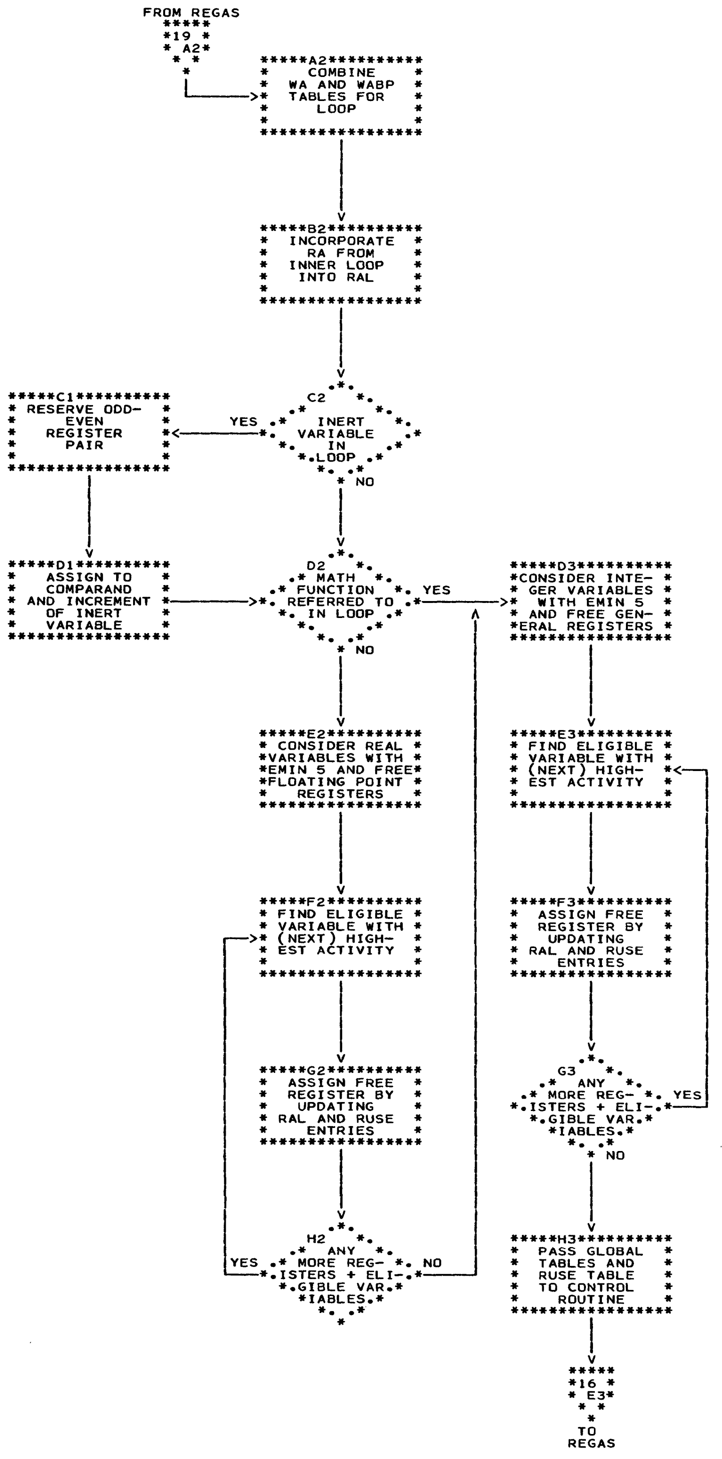

FROM REGAS
19 A2
A2 COMBINE WA AND WABP TABLES FOR LOOP
B2 INCORPORATE RA FROM INNER LOOP INTO RAL
C2 INERT VARIABLE IN LOOP
YES
NO
C1 RESERVE ODD-EVEN REGISTER PAIR
D1 ASSIGN TO COMPARAND AND INCREMENT OF INERT VARIABLE
D2 MATH FUNCTION REFERRED TO IN LOOP
YES
NO
D3 CONSIDER INTEGER VARIABLES WITH EMIN 5 AND FREE GENERAL REGISTERS
E2 CONSIDER REAL VARIABLES WITH EMIN 5 AND FREE FLOATING POINT REGISTERS
E3 FIND ELIGIBLE VARIABLE WITH (NEXT) HIGHEST ACTIVITY
F2 FIND ELIGIBLE VARIABLE WITH (NEXT) HIGHEST ACTIVITY
F3 ASSIGN FREE REGISTER BY UPDATING RAL AND RUSE ENTRIES
G2 ASSIGN FREE REGISTER BY UPDATING RAL AND RUSE ENTRIES
G3 ANY MORE REGISTERS + ELIGIBLE VARIABLES
YES
NO
H2 ANY MORE REGISTERS + ELIGIBLE VARIABLES
YES
NO
H3 PASS GLOBAL TABLES AND RUSE TABLE TO CONTROL ROUTINE
16 E3
TO REGAS

Chart 20. Text Updating (STXTR)

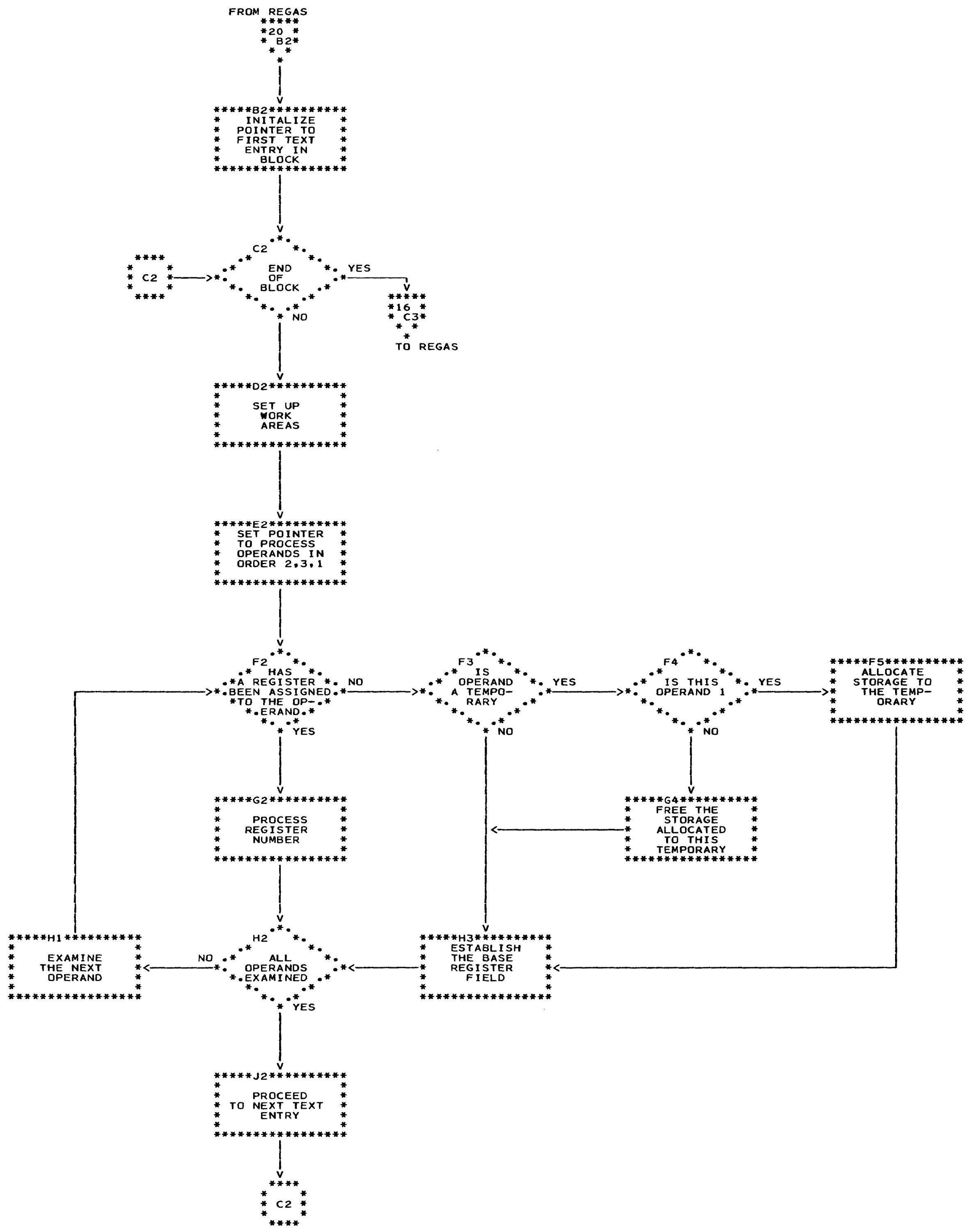

FROM REGAS
20 B2
B2 INITALIZE POINTER TO FIRST TEXT ENTRY IN BLOCK
C2
C2 END OF BLOCK
YES
NO
16 C3
TO REGAS
D2 SET UP WORK AREAS
E2 SET POINTER TO PROCESS OPERANDS IN ORDER 2,3,1
F2 HAS A REGISTER BEEN ASSIGNED TO THE OP-ERAND
NO
YES
F3 IS OPERAND A TEMPO-RARY
YES
NO
F4 IS THIS OPERAND 1
YES
NO
F5 ALLOCATE STORAGE TO THE TEMP-ORARY
G2 PROCESS REGISTER NUMBER
G4 FREE THE STORAGE ALLOCATED TO THIS TEMPORARY
H1 EXAMINE THE NEXT OPERAND
H2 ALL OPERANDS EXAMINED
NO
YES
H3 ESTABLISH THE BASE REGISTER FIELD
J2 PROCEED TO NEXT TEXT ENTRY
C2

Table 10. Criteria for Text Optimization

Process	Basic	Primary	Secondary
Common Expression Elimination (XPELIM)	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
Forward Movement (FORMOV)	Arithmetic or logical operator	Operand 1 unused in the loop	Operand 1, operand 2, operand 3 undefined below the text item
Backward Movement (BACMOV)	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 (REDUCE)	Additive operator; inert variable	Interaction of inert variable with additive or multiplicative operator	Function of absolute constants or stored constants
Constant Expression Reordering (AGGLUT)	Additive or multiplicative operator; constant operand	Interaction of additive or multiplicative operator with another	Function of absolute constants or stored constants

Table 11. Phase 20 Subroutine Directory

Subroutine	Function
ACCEPT	Performs final acceptance test on variables which are candidates for local register assignment.
AGGLUT	Controls constant expression reordering.
ALLCOR	Allocates main storage to temporaries when necessary during text updating.
BACMOV	Controls backward movement.
BAKT	Computes the loop number of each module block.
BASVAR	Assigns eminence values to base variables, and sets up composite MVF and MVS matrixes.
BIZX	Computes the proper MVX setting for each variable in each block of the module.
BKDMP	Printing routine for full register assignment.
BKP	Common block for local register assignment.
BKPAS	Controls local register assignment.
BLK	Common block for structural determination routines.
BLS	Computes the total size of each block in the module.
BLSDTA	Block data for branching optimization.
BSTRIP	Block data for branching optimization.
BSYONX	Identifies forward target (if any) of a loop and sets up composite MVX matrix.
CNT	Block data area for phase 20.
CXIMAG	Processes imaginary parts of complex functions during local register assignment.
DISCHK	Performs a displacement check on a subscript text items during local register assignment.
FCLT50	Performs special checks on text items whose function codes are less than 50.
FOLLOW	Determines if interfering block causes redefinition of a variable.
FORMOV	Controls forward movement.
FREE	Releases busy registers during overflow conditions (local assignment).
FWDPAS	Table-building routine for full register assignment.
FWDPS1	Determines if text operands are register candidates prior to local register assignment.
FWP	Common block for local register assignment.
GLOBAS	Assigns most active variables to registers across the loop.
GLOBS1	Provides (if necessary) loads and stores for variables globally assigned outside the loop.
GLS	Common block for global assignment.

GROUPA	Performs reordering of type 5 with type 5 entries.
GROUPB	Performs reordering of type 6 with type 5 entries.
GROUPC	Performs reordering of type 6 with type 6 entries.
GTBASE	Gets a base register for operands of text items during text updating.
HILOWS	Determines if an even-odd register pair is available for indexing.
INDTRY	Determines if an inert variable is valid for the entire loop.
INERT	Produces new inert text entries for strength reduction.
INVERT	Gets text pointers in a backward direction.
LOC	Block data for register assignment.
LPSEL	Controls sequencing of loops and passes control to text optimization and register assignment routines.
LYT	Determines which module blocks can be reached via RX branch instructions.
MBRAN	Controls alternation of the compare and test entry for strength reduction.
MRCLEN	Performs special checks on text items whose function codes are greater than 55.
NORMIZ	Builds type tables for use by strength reduction and constant reordering.
NPRFUN	Controls phase 20 printing.
OPT	Common block for phase 20.
PERTRY	Performs compile-time mode conversions.
PRELUD	Determines if block under consideration has a branch which transfers out of the loop.
PROP1	Processes operand 1 of text item being processed by local register assignment.
REDUCE	Controls strength reduction.
REG	Common block for register assignment.
REGAS	Controls full register assignment.
RELCOR	Releases temporary main storage so it can be reused.
SEARCH	Provides register loads upon entering the module.
SEG4	Computes size of prologues, epilogues, and entry code.
SETREG	Updates text items to reflect global register assignments.
SETUP	Performs initialization for each text item during local assignment.
SHARE	Determines if the register assigned to operand 2 or 3 can be assigned to operand 1 during local register assignment.
SPLRA	Assigns registers during basic register assignment.
SRPRIZ	Prints a flowchart indicating the structure of the module.
SSTAT	Sets status information for operands and base addresses of text entries.
STDMP	Printing routine for basic register assignment.

STX	Common block for text updating.
STXTR	Controls text updating.
SUBTRY	Checks conditions for elimination during backward movement.
TAGLOC	Determines new operators for constant expression reordering.
TARGET	Identifies the members of a loop and its back target.
TOPO	Computes the immediate back dominator of each block in the module.
TRNSFM	Performs special checks on text items whose function codes are in the range of 50 to 55 inclusive.
TYPLOC	Locates interactions of text entries for constant expression reordering.
XCHANG	Determines stored constants for common expression elimination.
XPELIM	Controls common expression elimination.
XPELOC	Locates common text entries in a local block during common expression elimination.
XPLACE	Performs manipulations for common expression elimination.
XSCAN	Performs local block scan for common expression elimination.
YCHANG	Determines stored constants for backward movement.
YPLACE	Performs manipulations for backward movement.
ZCHANG	Determines stored constants for forward movement.
ZPLACE	Performs manipulations for forward movement.

Table 12. Phase 20 Utility Subroutines

Subroutine	Function
CIRCLE	Examines composite vectors, or each local vector if necessary.
CLASIF	Classifies operands of the current text entry.
DELTEX	Deletes the current text entry by rechaining.
FILTEX	Fills text space according to the arguments.
GETDIC	Gets space for temporary cells.
GETDIK	Gets space for constants.
GETSPC	Gets space for new text item.
KORAN	Performs bit manipulation for text optimization.
LORAN	Updates composite MVS and MVF matrixes.
MODFIX	Adjusts text entry for possible mode change.
MOV	Common block for text optimization.
MOVTEX	Moves text entries by rechaining, and updates MVS and MVF vectors.
MOZ	Common block for text optimization.
OBTAIN	Obtains next local block for processing.
PARFIX	Changes parameter list to correspond to text replacements.
PERFOR	Performs combination of constants at compile time.
SUBACT	Performs replacement of operands with equivalent values.
SUBSUM	Replaces, if possible, operand values with equivalent values.
WRITEX	Printing routine for text optimization.
YSCAN	Performs local block scan for backward movement.
ZSCAN	Performs local block scan for forward movement.

Chart 21. Phase 25 (Initial Text Info Const)

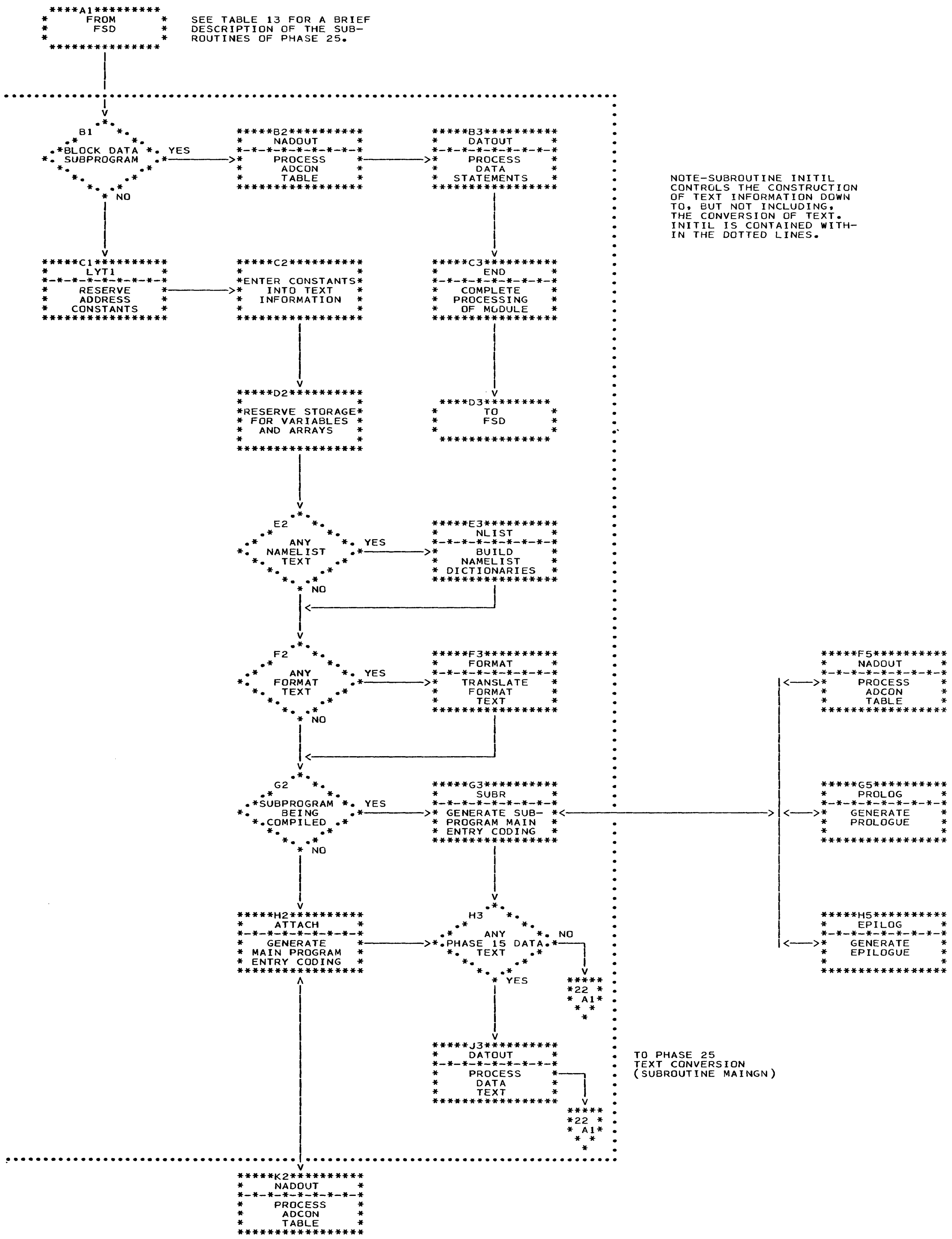

A1 FROM FSD
SEE TABLE 13 FOR A BRIEF DESCRIPTION OF THE SUB-ROUTINES OF PHASE 25.
B1 BLOCK DATA SUBPROGRAM
YES
NO
B2 NADOUT PROCESS ADCON TABLE
B3 DATOUT PROCESS DATA STATEMENTS
NOTE-SUBROUTINE INITIL CONTROLS THE CONSTRUCTION OF TEXT INFORMATION DOWN TO, BUT NOT INCLUDING, THE CONVERSION OF TEXT. INITIL IS CONTAINED WITH-IN THE DOTTED LINES.
C1 LYT1 RESERVE ADDRESS CONSTANTS
C2 ENTER CONSTANTS INTO TEXT INFORMATION
C3 END COMPLETE PROCESSING OF MODULE
D2 RESERVE STORAGE FOR VARIABLES AND ARRAYS
D3 TO FSD
E2 ANY NAMELIST TEXT
E3 NLIST BUILD NAMELIST DICTIONARIES
F2 ANY FORMAT TEXT
F3 FORMAT TRANSLATE FORMAT TEXT
F5 NADOUT PROCESS ADCON TABLE
G2 SUBPROGRAM BEING COMPILED
G3 SUBR GENERATE SUB-PROGRAM MAIN ENTRY CODING
G5 PROLOG GENERATE PROLOGUE
H2 ATTACH GENERATE MAIN PROGRAM ENTRY CODING
H3 ANY PHASE 15 DATA TEXT
22 A1
H5 EPILOG GENERATE EPILOGUE
J3 DATOUT PROCESS DATA TEXT
22 A1
TO PHASE 25 TEXT CONVERSION (SUBROUTINE MAINGN)
K2 NADOUT PROCESS ADCON TABLE

Chart 22. Phase 25 (Text Conversion)

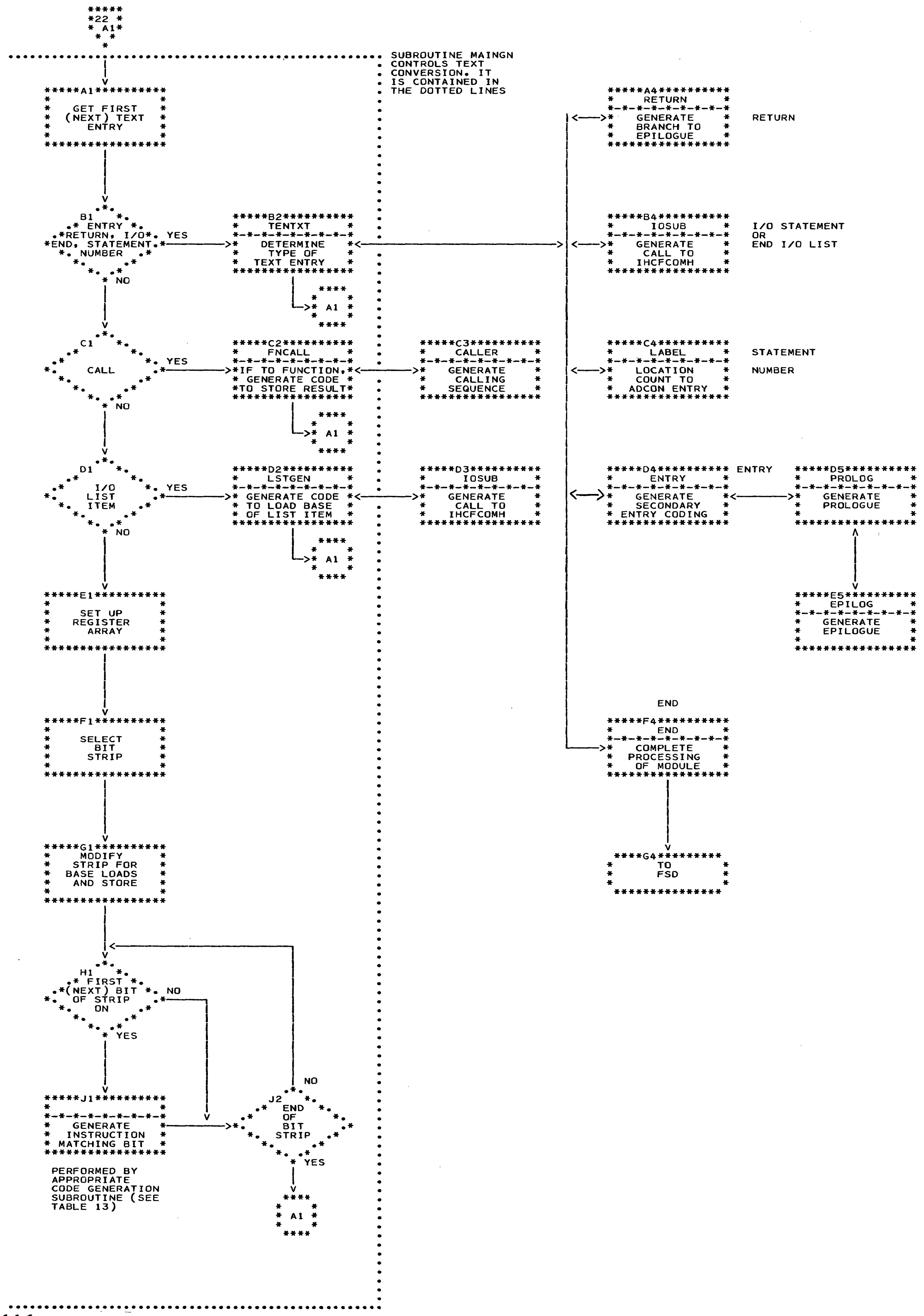

22 A1
SUBROUTINE MAINGN CONTROLS TEXT CONVERSION. IT IS CONTAINED IN THE DOTTED LINES
A1 GET FIRST (NEXT) TEXT ENTRY
B1 ENTRY, RETURN, I/O, END, STATEMENT NUMBER
YES
NO
B2 TENTXT DETERMINE TYPE OF TEXT ENTRY
A1
C1 CALL
YES
NO
C2 FNCALL IF TO FUNCTION, GENERATE CODE TO STORE RESULT
A1
C3 CALLER GENERATE CALLING SEQUENCE
D1 I/O LIST ITEM
YES
NO
D2 LSTGEN GENERATE CODE TO LOAD BASE OF LIST ITEM
A1
D3 IOSUB GENERATE CALL TO IHCFCOMH
E1 SET UP REGISTER ARRAY
F1 SELECT BIT STRIP
G1 MODIFY STRIP FOR BASE LOADS AND STORE
H1 FIRST (NEXT) BIT OF STRIP ON
NO
YES
J1 GENERATE INSTRUCTION MATCHING BIT
PERFORMED BY APPROPRIATE CODE GENERATION SUBROUTINE (SEE TABLE 13)
J2 END OF BIT STRIP
NO
YES
A1
A4 RETURN GENERATE BRANCH TO EPILOGUE
RETURN
B4 IOSUB GENERATE CALL TO IHCFCOMH
I/O STATEMENT OR END I/O LIST
C4 LABEL LOCATION COUNT TO ADCON ENTRY
STATEMENT NUMBER
D4 ENTRY GENERATE SECONDARY ENTRY CODING
ENTRY
D5 PROLOG GENERATE PROLOGUE
E5 EPILOG GENERATE EPILOGUE
END
F4 END COMPLETE PROCESSING OF MODULE
G4 TO FSD

Table 13. Phase 25 Subroutine Directory

Subroutine	Function
ABSGEN[1]	Generates instructions for ABS, IABS, and DABS in-line functions.
ADMDGN[1]	Generates instructions for the AMOD and DMOD in-line functions.
ATTACH	Generates main program entry coding.
BDATA	Initializes the masks and flags used by phase 25.
BITNFP[1]	Generates instructions for the BITON, BITOFF, and BITFLP in-line functions.
BRANCH[1]	Generates the instructions for all unconditional branching.
BRCOMB[1]	Generates instructions for computed GO TO operations.
BRCOMP[1]	Generates instructions for assigned GO TO operations.
BRLGL[1]	Generates instructions for text entries whose operator is a relational operator operating upon two operands or one operand and zero.
BTBF[1]	Generates instructions for branch true and branch false operations.
BXHCOM	Common data area used by phase 25.
CALLER	Generates calling sequences for CALLs (other than those to IHCFCOMH) and function references.
CGNDTA	Initializes the arrays used during code generation.
CMPLGN[1]	Generates instructions for the COMPL and LCOMPL in-line functions.
DATOUT	Processes phase 15 data text by entering into text information the initial data values at the appropriate variable locations.
DBLGEN[1]	Generates instructions for the DBLE in-line function.
DCLIST	Produces a listing of the address constants of the object module.
DIMGEN[1]	Generates instructions for the DIM and IDIM in-line functions.
DIVGEN[1]	Generates instructions for all half- and full-word integer division.
END	Completes the processing of the object module.
ENTRY	Generates subprogram secondary entry coding.
EPILOG	Generates the epilogues associated with a subprogram and its secondary entry points (if any).
FAZ25	Common data area used by phase 25.
FLTGEN[1]	Generates instructions for the FLOAT and DFLOAT in-line functions.
FNCALL	Generates the instructions to store the result returned by a function subprogram.
FORMAT	Translates FORMAT statements to a form acceptable to IHCFCOMH.
GOTOKK	Used by subroutine MAINGN to branch to the code generation subroutines.
IEKTLOAD	Builds ESD, TXT, RLD, and loader END records.
IEKWAG[1]	Generates the instructions to implement the ASSIGN statement.

INITIA	Interface between FSD and subroutine INITIL.
INITIL	Controls the construction of that portion of text information down to but not including text conversion.
INTMPY[1]	Generates instructions for all half- and full-word integer division.
IOSUB/ IOSUB2	Generate calling sequences for calls to IHCFCOMH.
LABEL	Processes statement numbers by entering the current value of the location counter into the address constant reserved for the statement number.
LBITTF[1]	Generates the instructions for the TBIT in-line function.
LDADDR[1]	Generates the instructions for all load address operations.
LDBGEN[1]	Generates the instructions for all load byte operations.
LGLNOT[1]	Generates the instructions for logical NOT operations.
LISTER	Produces a listing of the final compiler generated instructions.
LYT1	Reserves address constants for statement numbers.
MAINGN/ MANGN2	Control the text conversion process of Phase 25.
MINUS[1]	Generates the instructions for all subtraction operations.
MOD24[1]	Generates the instructions for the MOD24 in-line function.
MXMNGN[1]	Generates the instructions for the MAX2 and MIN2 in-line functions.
NADOUT	Enters the address constants developed during the compilation into text information.
NDORGN[1]	Generates the instructions for the AND and OR in-line functions.
NLIST	Builds the object-time namelist dictionaries.
NTFXGN[1]	Generates the instructions for the INT, IDINT, IFIX, and HFIX in-line functions.
PACKER	Packs the various parts of each instruction produced during code generation into a TXT record.
PLSGEN[1]	Generates the instructions for all addition operations and for real multiplication and division operations.
PROLOG	Generates prologues for subprograms and secondary entry points (if any).
RETURN	Processes the RETURN statement by generating a branch to the epilogue.
SHFT2[1]	Generates the instructions for all right- and left-shift operations.
SHFTRL[1]	Generates the instructions for the SHFTR and SHFTL in-line functions.
SIGNGN[1]	Generates the instructions for the SIGN, ISIGN, and DSIGN in-line functions.
STOPPR[1]	Generates character strings in calls to IHCFCOMH for STOP and PAUSE statements.
STRGEN[1]	Generates the instructions for all store operations.
SUBGEN[1]	Generates the instructions for subscript text entries.

SUBR	Generates subprogram main entry coding.
TENTXT	Controls the processing of END, RETURN, I/O, and ENTRY statements, statement numbers, and end of I/O list indicators.
TSTSET[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.
UNRGEN[1]	Generates the instructions for unary minus operations (e.g., A=-B).

[1]Code generation subroutine.

Chart 23. Phase 30 (IEKP30) Overall Logic

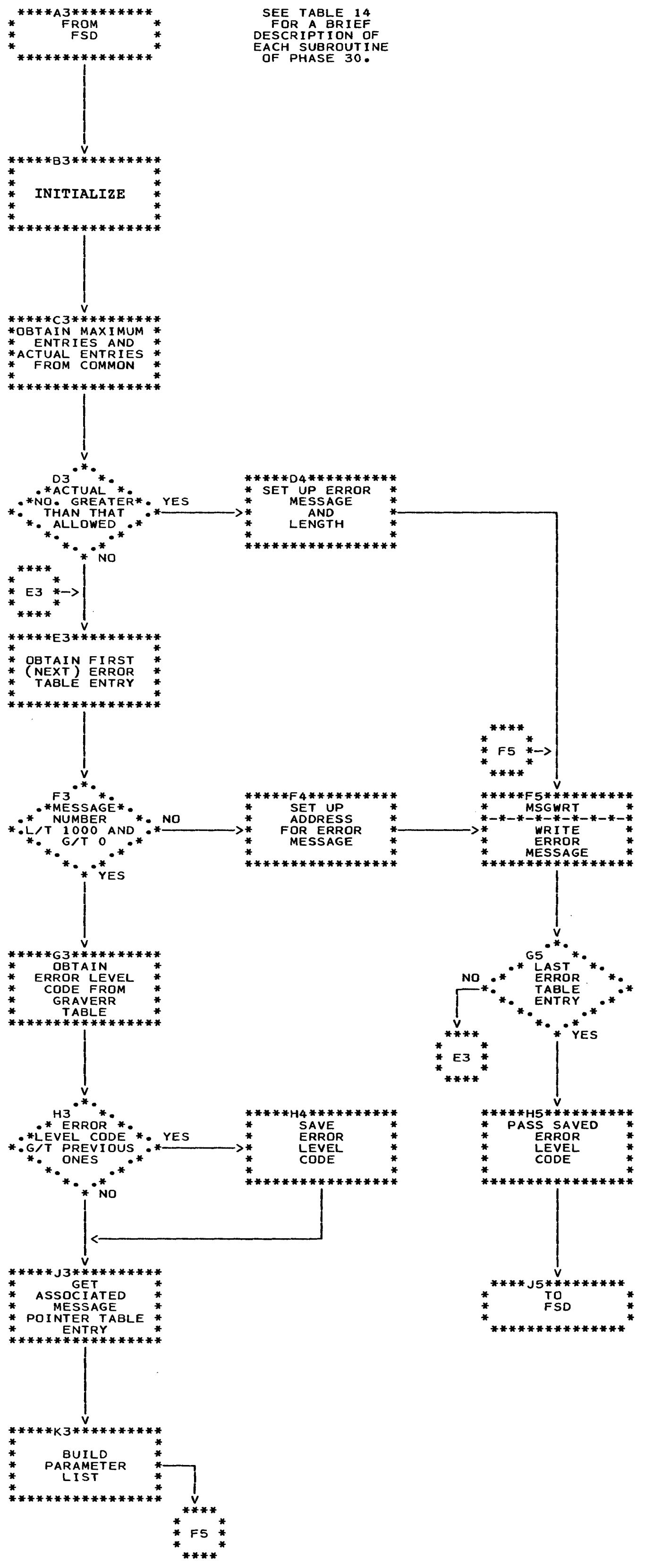

A3 FROM FSD
SEE TABLE 14 FOR A BRIEF DESCRIPTION OF EACH SUBROUTINE OF PHASE 30.
B3 INITIALIZE
C3 OBTAIN MAXIMUM ENTRIES AND ACTUAL ENTRIES FROM COMMON
D3 ACTUAL NO. GREATER THAN THAT ALLOWED
YES
NO
D4 SET UP ERROR MESSAGE AND LENGTH
E3
E3 OBTAIN FIRST (NEXT) ERROR TABLE ENTRY
F3 MESSAGE NUMBER L/T 1000 AND G/T 0
NO
YES
F4 SET UP ADDRESS FOR ERROR MESSAGE
F5
F5 MSGWRT WRITE ERROR MESSAGE
G3 OBTAIN ERROR LEVEL CODE FROM GRAVERR TABLE
G5 LAST ERROR TABLE ENTRY
NO
YES
E3
H3 ERROR LEVEL CODE G/T PREVIOUS ONES
YES
NO
H4 SAVE ERROR LEVEL CODE
H5 PASS SAVED ERROR LEVEL CODE
J3 GET ASSOCIATED MESSAGE POINTER TABLE ENTRY
J5 TO FSD
K3 BUILD PARAMETER LIST
F5

Table 14. Phase 30 Subroutine Directory

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

APPENDIX A: TABLES

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 15 illustrates the organization of the communication table.

Table 15. Communication Table (NPTR(2,35))

1		Pointer to 1-character symbol chain
2	Previous Classification code (phase 10)	Pointer to 2-character symbol chain
3	Options (e.g., SOURCE, MAP)	Pointer to 3-character symbol chain
4		Pointer to 4-character symbol chain
5	Displacement for temporary (phase 20)	Pointer to 5-character symbol chain
6	Maximum line count	Pointer to 6-character symbol chain
7	Reserved	Reserved
8	Type of text (phase 10)	Reserved
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 switch	Trace switch
12	Last error table entry	
13	GETCD 'END' card indicator	
14	Pointer to parameters	Pointer to 4-byte constant chain
15	Addr. const. entry number	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	Reserved	1,34 copied here by phase 20
19	Reserved	2,34 copied here by phase 20
20	Reserved	Reserved
21	Reserved	Pointer to common address constants
22	Pointer to dictionary entry for IBCOM	Next available error table entry
23	External function or CALL indicator	Pointer to end of statement number chain
24	Pointer to in-line function storage	Optimization switch
25		Pointer to common chain
26	Reserved	Pointer to equivalence chain
27	Pointer to literal constant chain	Pointer to data text chain
28	Instruction count	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	Adcon table number	Adcon entry number
35	Size of common	Delete/error switch

CLASSIFICATION TABLES

Classifying, a function of the preparatory subroutine (GETCD) of phase 10, involves the assignment of a code to each type of source statement. This code indicates to the DSPTCH subroutine which subroutine (either keyword or arithemtic) 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 16 and 17, 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 classifying process.

Table 16. Keyword Pointer Table

Character (1 word)	Number[1] (1 word)	Displacement[2] (1 word)
A	1	0
B	2	8
C	5	30
D	7	80
E	5	159
F	2	203
G	1	221
H	0	0
I	5	227
J	0	0
K	0	0
L	2	271
M	1	297
N	2	303
O	0	0
P	3	321
Q	0	0
R	5	342
S	3	384
T	2	413
U	0	0
V	0	0
W	1	432
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 17. Keyword Table

Number of Characters In Key Word Minus 1 (1 byte)	Key Word (1 word/character)	Classification Code (1 byte)
5	ASSIGN	1
8	BACKSPACE	2
8	BLOCKDATA	3
14	COMPLEXFUNCTION	4
7	CONTINUE	5
6	COMPLEX	6
5	COMMON	7
3	CALL	8
22	DOUBLEPRECISIONFUNCTION	10
14	DOUBLEPRECISION	11
8	DIMENSION	14
6	DISPLAY	15
4	DEBUG	16
3	DATA	17
1	DO	18
10	EQUIVALENCE	19
7	EXTERNAL	20
6	ENDFILE	21
4	ENTRY	22
2	END	23
7	FUNCTION	24
5	FORMAT	25
3	GOTO	27
14	INTEGERFUNCTION	28
7	IMPLICIT	29
6	INTEGER	30
1	IF(Arithmetic)	31
1	IF(Logical)	32
14	LOGICALFUNCTION	33
6	LOGICAL	34
3	MOVE	35

7	NAMELIST	36
5	NORMAL	37
4	PAUSE	38
4	PRINT	39
4	PUNCH	40
11	REALFUNCTION	41
5	REWIND	42
5	RETURN	43
3	READ	44
3	REAL	45
9	SUBROUTINE	46
8	STRUCTURE	47
3	STOP	48
7	TRACEOFF	49
6	TRACEON	50
4	WRITE	51

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 13 illustrates this concept.

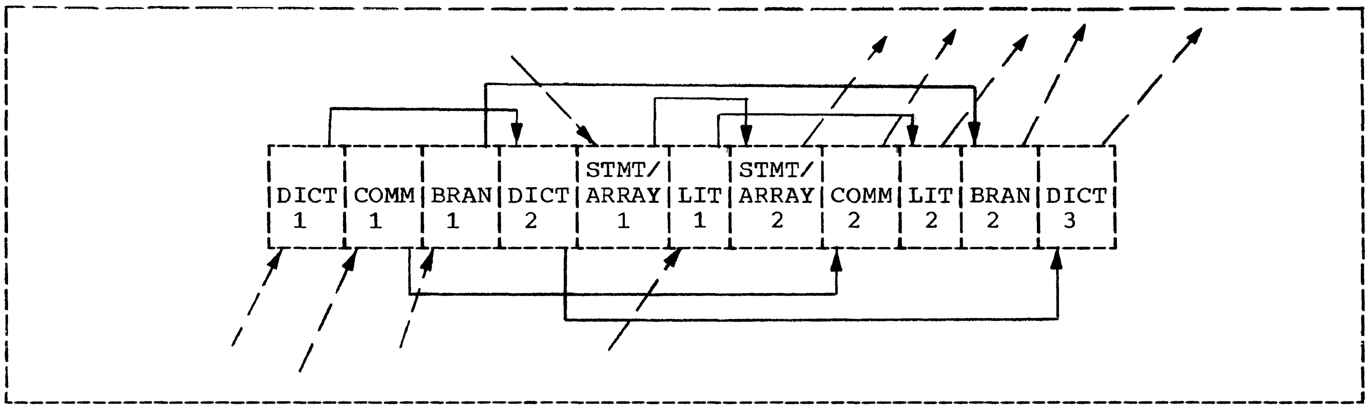


Figure 13. Information Table Chains

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 found, the item is not re-entered. 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 "binary tree" notation. This notation provides two chains, high and low (with a common starting point), for the entries describing variables of the same length or constants of the same size. The common starting point is the first entry placed into the information table for a variable of a particular length or a constant of a particular size. The following example illustrates the manner in which phase 10 employs the binary 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 common starting 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 common starting 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.

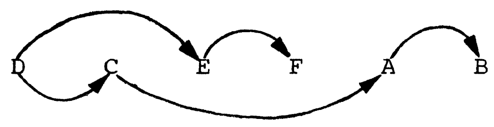
When the variable E is encountered, phase 10 carries out essentially the same procedure; however, because E is greater than D, phase 10 examines the high chain field of the entry for D. It is zero, which denotes the end of the chain. Phase 10 therefore places the dictionary entry for E into the next 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 common starting point for the chain. Because E 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 non-zero 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.

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



Note: The pointers from the top of one variable to the top of another variable represent high chain pointers. The pointers from the bottom of one variable to the bottom of another variable represent low chain pointers.

Figure 14. 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 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, and 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 is found by scanning the chain of dictionary entries for the variables in the common block until a zero common chain field is detected.) 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 scans the chain of dictionary entries constructed for the variables associated with the first mention of the common block name. When the last entry in the chain is found, it records in the common chain field of that entry a pointer to the dictionary entry for the new variable. 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.

Literal Constant Chain Operation

Phase 10 constructs the literal constant chain in the same manner as it constructs the statement number chain. It records a pointer to the first literal constant entry (refer to "Literal Table") it enters in the information table in the appropriate field of the communication table. For each other literal constant entry, phase 10 records a pointer to its location in the information table in the chain field of the previously developed literal constant entry, which is found by scanning the chain of such entries until a zero chain field is found.

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.

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 15.

Byte A usage field	(1 word)
Low chain field	(1 word)
Byte B usage field	(1 word)
High chain field	(1 word)
Mode/type field	(2 words)
P1 field	(1 word)
Byte C usage field (Used by phase 15)	(1 word)
Used by Phase 15	(1 word)
Used by Phase 15	(1 word)
Common chain field	(1 word)
Name field	(2 words)

Figure 15. Format of Dictionary Entry for Variable

Byte A Usage Field: This field is contained in a full word, the high-order three bytes of which are not used. This field indicates a portion of the characteristics of the variable for which the dictionary 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 16 indicates the function of each subfield in the byte A usage field.

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

Figure 16. Function of Each Subfield in the Byte A Usage Field of a Dictionary Entry for a Variable

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.

Byte B Usage Field: The byte B usage field is contained in a full word, the high-order three bytes of which are not used. 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 17 illustrates the function of each subfield in the byte B usage field.

Subfield	Function
Bit 0 'on'	variable is "call by value" parameter
Bit 1 'on'	variable is "call by name" parameter
Bit 2 'on'	variable is used as an argument
Bit 3 'on'	variable is used in NAMELIST statement
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 17. Function of Each Subfield in the Byte B Usage Field of a Dictionary Entry for a Variable

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.

Mode/Type Field: The mode/type field is divided into two subfields, each one word long. The first word (mode subfield) is used to indicate the mode of the variable (e.g., integer, real); the second word (type subfield) is 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 18 and 19).

Table 18. 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

Table 19. 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 (in text)	B
Dummy array (in dictionary)	B
Negative external function	C
Negative constant	D
Negative statement function	E

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

Common 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.

MODIFIC TIONS 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 15 processing. Only changes are indicated; * stands for unchanged.

Dictionary Entry for Variable After Dictionary Sorting: The format of a dictionary entry for a variable after the dictionary has been sorted during STALL is illustrated in Figure 18.

*	(1 word)
Freed by sorting	(1 word)
*	(1 word)
New chain field	(1 word)
*	(2 words)
*	(1 word)
*	(1 word)
*	(1 word)
*	(1 word)
*	(1 word)
*	(2 words)

Figure 18. Format of Dictionary Entry for Variable After Sorting

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 19.

*	(1 word)
Freed by sorting	(1 word)
*	(1 word)
New chain field	(1 word)
*	(2 words)
*	(1 word)
*	(1 word)
Displacement from start of common block (if variable is in common)	(1 word)
Pointer to common block name entry for block containing variable	(1 word)
*	(1 word)
*	(2 words)

Figure 19. Format of Dictionary Entry for Variable After Commom Block Processing

Dictionary Entry for Variable After PHAZ15 Processing: The format of a dictionary entry for a variable after PHAZ15 processing is illustrated in Figure 20.

*	(1 word)
Freed by sorting	(1 word)
*	(1 word)
New chain field	(1 word)
*	(2 words)
*	(1 word)
Coordinate number for variable	(1 word)
Displacement from start of common block (if variable is in common)	(1 word)
Pointer to common block name entry for block containing variable	(1 word)
*	(1 word)
*	(2 words)

Figure 20. Format of Dictionary Entry for Variable After PHAZ15 Processing

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 21.

*	(1 word)
Pointer to entry containing pointer to the address constant for the variable	(1 word)
*	(1 word)
New chain field	(1 word)
*	(2 words)
*	(1 word)
Coordinate number for variable	(1 word)
Displacement from associated address constant	(1 word)
Pointer to common block name entry for block containing variable	(1 word)
*	(1 word)
*	(2 words)

Figure 21. 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 22.

Byte A usage field	(1 word)
Low chain field	(1 word)
Byte B usage field	(1 word)
High chain address field	(1 word)
Mode/type field	(2 words)
Not used	(1 word)
Byte C usage field (used by phase 15)	(1 word)
Used by phase 15	(1 word)
Constant field	(4 words)

Figure 22. Format of Dictionary Entry for Constant

The byte A usage, low chain, byte B usage, high chain, and mode/type fields of a dictionary entry for a constant contain the same information as a dictionary entry for a variable.

Constant Field: The field contains the binary equivalent of the constant for which the dictionary entry was constructed.

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

Dictionary Entry for Constant After Dictionary Sorting: The format of a dictionary entry for a constant after the dictionary has been sorted is illustrated in Figure 23.

*	(1 word)
Freed by sorting	(1 word)
*	(1 word)
New chain field	(1 word)
*	(2 words)
*	(1 word)
*	(1 word)
*	(1 word)
*	(4 words)

Figure 23. Format of Dictionary Entry for Constant After Sorting

Dictionary Entry for Constant After PHAZ15 Processing: The format of a dictionary entry for a constant after the processing of PHAZ15 is illustrated in Figure 24.

*	(1 word)
Freed by sorting	(1 word)
*	(1 word)
New chain field	(1 word)
*	(2 words)
*	(1 word)
Coordinate number for constant	(1 word)
*	(1 word)
*	(4 words)

Figure 24. Format of Dictionary for Constant After PHAZ15 Processing

Dictionary Entry for Constant After Relative Address Assignment: The format of a dictionary entry for a constant after the relative address assignment processes is complete is illustrated in Figure 25.

*	(1 word)
Pointer to entry containing pointer to the address constant for the constant	(1 word)
*	1 WORD)
New chain field	(1 WORD)
*	(2 words)
*	(1 word)
Coordinate number for constant	(1 word)
Displacement from associated address constant	(1 word)
*	(4 words)

Figure 25. Format of Dictionary Entry for Constant After Relative Address Assignment

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 26.

Byte A usage field	(1 word)
Chain field	(1 word)
Not used	(1 word)
Pointer field	(1 word)
Byte B usage field	(1 word)
Image field	(1 word)
Used by Phase 20	(1 word)
Used by Phase 20	(1 word)
Used by Phase 15	(1 word)
Used by Phase 15	(1 word)
Used by Phase 20	(1 word)
Used by Phase 15	(1 word)
Not used	(1 word)

Figure 26. Format of a Statement Number Entry

Byte A Usage Field: This field is contained in a full word, the high-order three bytes of which are not used. This field indicates a portion of the characteristics of the statement number for which the entry was created. The bytes A usage field is divided into eight subfields, each of which is one bit long. The bits are numbered from 0 through 7. Figure 27 indicates the function of each subfield of this field.

Subfield	Function
Bit 0 'on'	statement number defined
Bit 1 'on'	statement number referenced
Bit 2 'on'	referenced 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 27. Function of Each Subfield in the Byte A Usage Field 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.

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

Byte B Usage Field: This field is contained in a full word, the high-order three bytes of which are not used. 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 28 indicates the function of each subfield in the byte B usage field.

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 2-6	not used
Bit 7 'on'	statement number is used in a computed GO TO statement

Figure 28. Function of Each Subfield in the Byte B Usage Field of a Statement Number Entry

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 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. Figure 29 illustrates the format of a statement number entry after the processing of phases 15, 20 and 25. Only changes are indicated; * stands for unchanged. The phase making the indicated change is specified within parentheses.

*	(1 word)
New chain field (phase 15)	(1 word)
*	(1 word)
Address constant pointer field (phase 20 or phase 25)	(1 word)
*	(1 word)
*	(1 word)
Loop number field (phase 20)	(1 word)
Back dominator field (phase 20)	(1 word)
Forward connection field (ILEAD) (phase 15)	(1 word)
Backward connection field (JLEAD) (phase 15)	(1 word)
Block status field (phase 20)	(1 word)
Text pointer field (phase 15)	(1 word)
*	(1 word)

Figure 29. Format of Statement Number Entry After the Processing of Phases 15, 20, and 25

New Chain Field: The new chain field (after phase 15 processing) 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. (Phase 15 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.

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 branchig 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").

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 is set up and used by phase 20.

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.

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.

Block Status Field: The block status field is contained in a full word, the low-order three bytes of which are not used. This field indicates the status of the text block associated with the statement number entry under consideration. The block status field is divided into eight subfield, each of which is one bit long. The bits are numbered 25 through 32. Figure 30 indicates the function of each subfield in the block status field.

Subfield	Function
Bit 25 Bit 26	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 27 'on'	the associated block exits from a loop
Bit 28 'on'	the associate block is a fork (i.e., it has two or more forward connections)
Bit 29	same as bits 25 and 26
Bit 30 'on'	the associated block is in the current loop
Bit 31 'on'	the associated block has been completely processed along the complete-optimized path
Bit 32 'or'	the associated block is an entry block

Figure 30. Function of Each Subfield in the Block Status Field

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.

DIMENSION ENTRY FORMAT: The format of the dimension entries constructed by phase 10 is illustrated in Figure 31.

Dimension number field	(1 word)
Not used	(1 word)
Array size field	(1 word)
Not used	(1 word)
Element length field	(1 word)
Second dimension factor field	(1 word)
Third dimension factor field	(1 word)
Fourth dimension factor field	(1 word)
Fifth dimension factor field	(1 word)
Sixth dimension factor field	(1 word)
Seventh dimension factor field	(1 word)
Pointer to last subscript parameter	(1 word)
Not used	(1 word)

Figure 31. Format of Dimension Entry

Dimension Number Field: The dimension number field contains the number of dimensions (1 through 7) of the associated array.

Array Size Field: The array size field contains either the total size of the associated array or zero, if the array has variable dimensions.

Element Length Field: The element length field contains the length of each element (first dimension factor) in the associated array.

Second Dimension Factor Field: The field contains either a pointer to the dictionary entry for the second dimension factor, which has a value of D1*L, or a pointer to the dictionary entry for the first subscript parameter used to dimension the associated array, if that array has variable dimensions.

Third Dimension Factor 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 is has a single dimension.

Fourth Dimension Factor 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 a variable dimensions. This field is not used if the associated array has fewer than three dimensions.

Fifth Dimension Factor 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.

Sixth Dimension Factor 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.

Seventh Dimension Factor 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 32.

Character number field	(1 word)
Chain field	(1 word)
Not used	(1 word)
P1 field	(1 word)
Not used	(1 word)
Used by phase 15	(1 word)
Name field	(2 words)
Not used	(5 words)

Figure 32. Format of a Common Block Name Entry

Character Number Field: The character number field contains the number of characters in the common block name.

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.

Name Field: The name field contains the name (right-justified) of the common block for which this common block name entry was constructed.

MODIFICATIONS TO COMMON BLOCK NAME ENTRIES: During compilation, certain fields of common block name entries may be modified. Figure 33 illustrates the format of a common block name entry after common block processing by STALL, the first segment of phase 15. Only changes are indicated; * stands for unchanged.

*	(1 word)
*	(1 word)
*	(1 word)
*	(1 word)
*	(1 word)
Total size of common block	(1 word)
*	(2 words)
*	(5 words)

Figure 33. 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 35.

Number field	(1 word)
Chain field	(1 word)
Not used	(1 word)
P1 field	(1 word)
Not used	(1 word)
Used by phase 15	(1 word)
Not used	(7 words)

Figure 35. Format of an Equivalence Group Entry

Number Field: The number field contains the number of variables being equivalenced in this equivalence group.

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.

P1 Field: The P1 field contains a pointer to the equivalence variable entry for the first variable in the equivalence group.

MODIFICATIONS TO EQUIVALENCE GROUP ENTRIES: During compilation, certain fields of equivalence group entries may be modified. Figure 36 illustrates the format of an equivalence group entry after equivalence processing by STALL, the first segment of phase 15. Only changes are indicated; * stands for unchanged.

*	(1 word)
*	(1 word)
*	(1 word)
*	(1 word)
*	(1 word)
Pointer to the "head" of the equivalence group	(1 word)
*	(7 words)

Figure 36. 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 37.

Used by phase 15	(1 word)
Offset field	(1 word)
Not used	(1 word)
P1 field	(1 word)
Not used	(1 word)
Chain field	(1 word)
Not used	(7 words)

Figure 37. Format of Equivalence Variable Entry

Offset Field: The offset field contains the displacement of this variable from the first element in the equivalence group.

P1 Field: The P1 field contains a pointer to the dictionary entry for this equivalence variable.

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.

MODIFICATIONS TO EQUIVALENCE VARIABLE ENTRIES: During compilation, certain fields of equivalence variable entries may be modified. Figure 38 illustrates the format of an equivalence variable entry after equivalence processing by STALL, the first segment of phase 15. Only changes are indicated; * stands for unchanged.

Null indicator	(1 word)
Displacement of variable from group "head"	(1 word)
*	(1 word)
*	(1 word)
*	(1 word)
*	(1 word)
*	(7 words)
The null indicator indicates to the relative address assignment portion of phase 15 that main storage has been previously allocated to this variable. This implies that the variable: (1) is also in common, or (2) appears in more than one equivalence group.	

Figure 38. 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 39.

Length field	(1 word)
Used by phase 15	(1 word)
Not used	(1 word)
Used by phase 15	(1 word)
Not used	(1 word)
Chain field	(1 word)
Literal constant field	(1-255 words)

Figure 39. Format of Literal Constant Entry

Length Field: The length field contains the length (in bytes) of the literal constant.

Chain Field: The chain field is used to maintain linkage between the various literal constant entries. It contains a pointer to the next literal constant entry.

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 words (1 character/word, 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 40 illustrates the format of a literal constant entry after relative address assignment by CORAL, the third segment of phase 15. Only changes are indicated; * stands for unchanged.

*	(1 word)
Pointer to entry containing pointer to the address constant for the literal constant	(1 word)
*	(1 word)
Displacement from associated address constant	(1 word)
*	(1 word)
*	(1 word)
*	(1-255 words)

Figure 40. 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 41.

Length field	(1 word)
Literal data field	(1-255 words)

Figure 41. 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 words (1 character/word, 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 upon encounter of 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 42.

Indicator field	(1 word)
Used by phase 25	(1 word)
Not used	(1 word)
Chain field	(1 word)
Not used	(1 word)
P1 field	(1 word)
Used by phase 25	(1 word)
Not used	(6 words)

Figure 42. Format on Initial Branch Table Entry

Indicator Field: The indicator field is non-zero 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 statement number for the compiler-generated statement number for the fall through statement.

MODIFICATIONS TO INITIAL BRANCH TABLE ENTRIES: During compilation certain fields of initial branch tabel entries may be modified. Figure 43 illustrates the format of an initial branch table entry after the processing of phase 25 is complete. Only changes are indicated; * stands for unchanged.

*	(1 word)
Pointer to address constant reserved for fall through statement number	(1 word)
*	(1 word)
*	(1 word)
*	(1 word)
*	(1 word)
Relative address of statement associated with fall through statement number	(1 word)
*	(6 words)

Figure 43. 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 illustrated in Figure 44.

Indicator field	(1 word)
Not used	(1 word)
Not used	(1 word)
Chain field	(1 word)
Not used	(1 word)
Pl field	(1 word)
Used by phase 25	(1 word)
Not used	(6 words)

Figure 44. Format of Standard Branch Table Entry

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.

Pl Field: The Pl 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 45 illustrates the format of a standard branch table entry after the processing of phase 25 is complete. Only changes are indicated; * stands for unchanged.

*	(1 word)
*	(1 word)
*	(1 word)
*	(1 word)
*	(1 word)
*	(1 word)
Relative address of statement associated with this statement number	(1 word)
*	(6 words)

Figure 45. Format of Standard Branch Table Entry After Phase 25 Processing

SUBPROGRAM TABLE

The subprogram table (referred to as IFUNT or IFUNTB) 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 establish subprogram/argument compatability. That is, phase 15 changes subprogram names (if necessary) so that the referenced subprogram or in-line routine is made to agree with the mode of the argument(s) to it. For example, if the FORTRAN programmer references the MOD in-line routine, and if the argument to be operated upon is of real mode, phase 15 replaces the reference to MOD with a reference to AMOD to ensure argument comparability[1].

Each entry in the subprogram table (see Table 19) contains three fields: usage (4 bytes), mode (2 bytes), and name (6 bytes).

Usage Field: For an in-line routine, the usage field contains an indication of the mode of the result returned from it (see Table 18). For a subprogram, the usage field is initially zero. If a subprogram is referred to in the source module (either explicitly when the subprogram referred to agrees with the mode of the argument to be operated upon, or implicitly either when the subprogram referred to is changed to ensure compatability or when exponentiation or complex multiplication and division operation are converted to a function reference), the arithmetic translator sets on the high order bit of the usage field in the entry for that subprogram. The relative address assignment portion of phase 15 interrogates in the high-order bit in the usage field for each subprogram. If on, an address constant is reserved for the subprogram, and a pointer to that address constant is placed into the usage field of the entry for that subprogram.

Mode Field: The mode field contains an indication of the mode of the arguments to the subprogram (see Table 18).

Name Field: The name field contains the name of the subprogram, right-justified.

---

[1]This process is called automatic typing.

Table 20. Subprogram Table

Index	Usage	Mode	Name
1		8	CDABS
2		9	CABS
3		6	DEXP
4		7	EXP
5		8	CDEXP
6		9	CEXP
7		6	DSIN
8		7	SIN
9		8	CDSIN
10		9	CSIN
11		6	DCOS
12		7	COS
13		8	CDCOS
14		9	CCOS
15		6	DSQRT
16		7	SQRT
17		8	CDSQRT
18		9	CSQRT
19		0	LOG
20		6	DLOG
21		7	ALOG
22		8	CDLOG
23		9	CLOG
24		0	LOG10
25		6	DLOG10
26		7	ALOG10
27		8	CDLG10
28		9	CLOG10
29		6	DATAN
30		7	ATAN
31		6	DATAN2
32		7	ATAN2
33		6	DSINH
34		7	SINH
35		6	DCOSH
36		7	COSH
37		6	DTANH
38		7	TANH
39		6	DTAN
40		7	TAN
41		6	DCOTAN
42		7	COTAN
43		6	DARSIN
44		7	ARSIN
45		6	DARCOS
46		7	ARCOS
47		6	DERF
48		7	ERF
49		6	DERFC
50		7	ERFC
51		6	DGAMMA
52		7	GAMMA
53		6	DLGAMA
54		7	ALGAMA
55		0	LGAMA
56			
57			
58			
59			
60		5	AMAX0
61		0	MAX
62		5	MAX0
63		6	DMAX1
64		7	AMAX1
65		7	MAX1
66		5	AMIN0
67		0	MIN
68		5	MIN0
69		6	DMIN1
70		7	AMIN1
71		7	MIN1
72		5	FIXPI#
73		6	FDXPD#
74		7	FRXPR#
75		6	FDXPI#
76		7	FRXPI#
77		8	FCDXI#
78		9	FCXPI#
79		8	CDDVD#
80		9	CDVD#
81		8	CDMPY#
82		9	CMPY#
83			MAX2#
84			MIN2#
85	7	7	DIM
86	5	5	IDIM
87	6	6	DMOD
88	5	5	MOD
89	7	7	AMOD
90	6	6	DSIGN
91	7	7	SIGN
92	5	5	ISIGN
93	6	6	DABS
94	7	7	ABS
95	5	5	IABS
96	6	6	IDINT
97	7	7	AINT
98	5	7	INT
99	4	7	HFIX
100	5	7	IFIX
101	6	5	DFLOAT
102	7	5	FLOAT
103	6	7	DBLE
104	0		BITON
105	0		BITOFF
106	0		BITFLP
107	7		AND
108	7		OR
109	7		COMPL
110	0		MOD24
111	3		LCOMPL
112	3		SHFTL
113	3		SHFTR
114	3		TBIT
115	3		LAND
116	3		LOR
117	3		LXOR
118			
119	5		ADDR
120	7	6	SNGL
121	7	9	REAL
122	7	9	AIMAG
123	8	6	DCMPLX
124	9	7	CMPLX
125	8	8	DCONJG
126	9	9	CONJG

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 21), global assignment tables (refer to table 22), 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.

Table 21. Local Assignment Tables

Name	Function	Origin[1]
J	Serves as index to TXP, BVR, BVRA, BVA.	FWDPAS
TXP	Gives the storage location of the text item associated with each value of J.	FWDPAS
BVR	Contains the MCOORD value associated with operand 1 of the text item represented by J.	FWDPAS
BVRA	Indicates the register locally assigned to the quantity represented by J.	BKPAS
BVA	Represents the activity within the block of the quantity represented by J; also contains indicator bits describing the quantity.	FWDPAS
WJ[2]	Indicates whether a variable is eligible for local assignment. Indexed via the MCOORD values obtained from BVR.	FWDPAS

[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 22. 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
RA	Indicates the number of the first register globally assigned to the variable represented by the MCOORD value; provides continuity in global assignment from inner to outer loops.	GLOBAS
RAL	Indicates the register globally assigned to the variable represented by the MCOORD value.	GLOBAS
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
WABP	Indicates the activity of base variables. Calculated in the same manner as the WA table.	FWDPAS

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.

If the contents of TRUSE(i) and RUSE(i) is equal to zero, then register i is available for assignment. If the value contained in TRUSE(i) or RUSE(i) is between 2 and 128, inclusive, then the register i is assigned to the variable whose MCOORD value is equal to the contents of TRUSE(i) or RUSE(i). If the contents of TRUSE(i) or RUSE(i) has a value between 252 and 255, register i is unavailable for assignment and is reserved for special use (see next paragraph).

Register Use Considerations: Registers 15 and 14 are not available for use by register assignment. They are reserved, and used for branching during the execution of the object module resulting from the compilation.

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 phase 25, "Initialization Instructions"). This register is also used to reference the adcon table.

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").

OPERATOR TABLE

The operator table (see Table 23) is used in the text-optimization process of constant expression recording. The operator table indicates the operators in the text entries that results from the application of constant expression reordering on a candidate pair of text entries.

Table 23. Operator Table

Group	Argument 1 (Definition)	Argument 2 (Use)	Function 1 (Constant Result)	Function 2 (New Definition)	Function 3 (Reordered Expression)
A	*	*	*	New Definition Not Required	*
	*	/	/		*
	*	⊣	⊣		⊣
	/	*	⊣		*
	/	/	*		/
	/	⊣	*		⊣
	⊣	*	*		⊣
	⊣	/	/		⊣
	⊣	⊣	⊣		*
B	+	*	*	*	+
	+	/	/	/	*
	-	*	*	*	-
	-	/	/	/	-
	←	*	*	*	←
	←	/	/	/	←
C	+	+	+	New Definition Not Required	+
	+	-	-		-
	+	←	←		←
	-	+	←		+
	-	-	+		-
	-	←	+		←
	←	+	+		←
	←	-	-		←
	←	←	←		+

Note: To accommodate non-communicative operations, the operators ← Subtract Reverse and ⊣ Divide Reverse have been ~~introduced~~.

egad!

146

NAMELIST DICTIONARIES

Namelist dictionaries are developed by phase 25 for the NAMELIST statements appearing in the source module. These dictionaries provide IHCFCOMH with the information required to implement READ/WRITE statements using NAMELISTs. The namelist dictionary constructed by phase 25 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 46.

Name field	(2 words)

Figure 46. 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 47.

Name field		(2 words)
Address field		(1 word)
Item Type field (1 byte)	Mode field (1 byte)	Not used (2 bytes)

Figure 47. Format of Namelist Variable Entry

Name Field: The name field contain 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 48.

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 48. 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 non-zero 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 non-zero.

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 (DI*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 to the error table. Each error table entry consists of 2 one-word fields. The first field contains either the internal statement number for the statement in which the error occurred or a pointer to the dictionary entry for a symbol that is in error (e.g., a variable that is incorrectly used in an EQUIVALENCE statement); 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 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 referenced 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 five types of intermediate text produced by phase 10:

- Normal text - the operator-operand pair representations of source statements other than DATA, NAMELIST, 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.
- 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: The intermediate text representations are comprised of individual text entries. Each intermediate 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 format text chain consists of linkages between the individual intermediate text representation 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 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 49 illustrates the format of an intermediate text entry constructed by phase 10.

Adjective code field	(1 word)	operator
Chain field	(1 word)	
Mode field	(1 word)	
Type field	(1 word)	
Pointer field	(1 word)	operand

Figure 49. 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 24). 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.

Table 24. 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
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		GOTO, and implied branches
193		BLOCK DATA
205		DATA
208		SUBROUTINE, FUNCTION, or ENTRY
209		FORMAT
210		End of I/O list
211		CONTINUE
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
226		READ using NAMELIST
230		I/O end-of-file parameter
231		I/O error parameter
232		BLANK
233	RET	RETURN
234	STOP	STOP
235		PAUSE
238		ASSIGN
241		Arithmetic assignment statement
243		Arithmetic IF
244		Relational IF
245	NDOIF	End of DO 'IF'
246		CALL
247	LIST	I/O or NAMELIST list item
248		NAMELIST
249	END	END
250		Computed GOTO
251		I/O unit number
252		FORMAT

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 17 and 18).

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.

Examples of Phase 10 Intermediate Text

An example of each type of phase 10 text (normal, data, namelist, 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 50.

Adjective Code	Chain	Mode	Type	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 word	1 word	1 word	1 word	1 word

[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 50. 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 51.

Adjective Code	Chain	Mode	Type	Pointer
DATA		0	0	→ To text for next DATA statement
0		0	0	ISN
0		Real	Scalar	→ A
,		Real	Scalar	→ B
/		Real	Constant	→ 2.1
,		Literal	Constant	→ 3HABC
/		Real	Scalar	→ C
,		Real	Scalar	→ D
/		Real	Constant	→ 1.
,		Real	Constant	→ 1.
/	0	0	0	0
1 word	1 word	1 word	1 word	1 word

Figure 51. 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 52.

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

Figure 52. Phase 10 Namelist 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 53.

Pointer Code	Chain	Mode	Type	Pointer
Statement number definition		Statement number	0	5
FORMAT		0	0	To text for next FORMAT statement
(2H0	A,A6	//5X	,3(I	4,E1
2.5,	3F12	.3,'	ABC'	))ǂ[1]
1 word	1 word	1 word	1 word	1 word
[1]Group mark.				

Figure 53. 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 54.

Adjective Code	Chain	Mode	Type	Pointer
(		0	0	1
+		Real	Scalar	→ D
*		0	0	2
*		Real	Scalar	→ E
/		0	0	3
)		0	0	Number of dummy arguments
End mark	0	0	0	0
1 word	1 word	1 word	1 word	1 word

Figure 54. 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 55.

Indicator field	(1 word)
Chain field	(1 word)
P1 field	(1 word)
P2 field	(1 word)
Offset field	(1 word)
Number field	(1 word)

Figure 55. 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 contained in a full word, the high-order three bytes of which are not used. The indicator field is divided into eight subfields, each of which is one bit long. The bits are numbered from 0 through 7. Figure 56 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 56. 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 exits.

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 57.

Chain field	(1 word)
Operator field	(1 word)
P1 field	(1 word)
Block size field	(1 word)
Indicator field	(1 word)
P2 field	(1 word)
Use vector field (MVF)	(4 words)
Definition vector field (MVS)	(4 words)
Busy-on-exit Vector field (MVX)	(4 words)

Figure 57. 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 25).

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

Block Size Field: The block size field contains the number of text entries within the block (started by the statement number for which the current text entry is made).

Table 25. 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
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
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		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
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
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
78	BITOFF	BITOFF in-line routine
78	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)
193		BLOCK DATA
200		COMMON
201		EQUIVALENCE
202		EXTERNAL
205		DATA
208		FUNCTION
209		FORMAT
210		END I/O
211		CONTINUE
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		Statement function
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

Indicator Field: The indicator field is contained in a full word, the high-order three bytes of which are not used. 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 25 through 32. Figure 58 indicates the function of each subfield in the indicator field.

Subfield	Function
Bits 25-28	not used
Bit 29 'on'	associated block contains an I/O operation
Bit 30 'on'	associated block contains a reference to a library function
Bit 31	not used
Bit 32 'on'	associated block contains an abnormal function reference

Figure 58. Function of Each Subfield in Indicator Field of Statement Number Text Entry

<u>P2 Field:</u> The P2 field contains a pointer to the last intermediate text entry within the block.

<u>Use Vector Field (MVF):</u> 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 phase 15, 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.

<u>Definition Vector Field (MVS):</u> The definition vector field is used to indicate which variables are defined in a block. Variables and constants, as they are encountered by Phase 15, 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.

<u>Busy-On-Exit Vector Field (MVX):</u> The busy-on-exit vector field in phase 15 indicates which variables are <u>not</u> 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 phase 15, 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 <u>not</u> 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.

<u>Standard Text</u>

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 59.

Chain field (1 word)				
Operator field (1 word)				
P1 field (1 word)				
P2 field (1 word)				
P3 field (1 word)				
Not used (bits 0-1)	Used by phase 20 (bits 2-13)	Not used (bits 14-25)	S field (bit 26)	Mode field (bits 27-31)
Not used (bits 0-7)	Used by phase 20 (bits 8-31)			
Displacement field (1 word)				

Figure 59. Format of a Standard Text Entry

<u>Chain Field:</u> The chain field is used to maintain the linkage between the various intermediate text entries. It contains a pointer to the next text entry.

<u>Operator Field:</u> The operator field contains an internal operation code (numeric) that indicates either the nature of the statement or the operation to the performed (see Table 25).

<u>P1 Field:</u> 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.

<u>P2 Field:</u> The P2 field contains either a pointer to the dictionary entry for operand 2 of the text entry, a pointer to an IFUNTB entry, or zero (0) if operand 2 does not exist.

<u>P3 Field:</u> 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

Table 26. Meanings of Bits in Mode Field of Standard Text Entry

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)

actual constant (for shifting operations), or zero (0) if operand 3 does not exist.

S Field: The S field indicates whether or not a text entry is involved in a subscript computation. (If the S bit is on (1), the text entry is part of a subscript computation.)

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 meanings of the bits in the mode field are given in Table 26.

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 60.

Chain field [1]						(1 word)
Operator field [1]						(1 word)
P1 field [1]						(1 word)
P2 field [1]						(1 word)
P3 field [1]						(1 word)
Not used (bits 0-1)	Status field (bits 2-13)	Not used (bits 14-25)	S field[1] (bit 26)	Mode field[1] (bits 27-31)		
Not used (bits 0-7)	R1 field (bits 8-11)	B1 field (bits 12-15)	R2 field (bits 16-19)	B2 field (bits 20-23)	R3 field (bits 24-27)	B3 field (bits 28-31)
Displacement field[1]						(1 word)
[1]The chain field, mode field, operator field, P1 field, P2 field, P3 field, S field, and displacement field are as defined in a phase 15 standard text entry. (Phase 20 does not alter these fields.)						

Figure 60. Format of Phase 20 Text Entry

<u>R1, R2, and R3 Fields:</u> 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.

<u>B1, B2, and B3 Fields:</u> 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.

<u>Status Field:</u> The status field is composed of 12 bits that are 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 29.

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 25.

Table 27. Status Field Bits and Their Meanings

Operand/ Base Address	Bit	Meaning
Operand 2 base address status	2	0 - base address in storage 1 - base address in register
	3	0 - do not retain base address in register 1 - retain base address in register
Operand 3 base address status	4	0 - base address in storage 1 - base address in register
	5	0 - do not retain base address in register 1 - retain base address in register
Operand 2 status	6	0 - operand in storage 1 - operand in register
	7	0 - do not retain operand in register 1 - retain operand in register
Operand 3 status	8	0 - operand in storage 1 - operand in register
	9	0 - do not retain operand in register 1 - retain operand in register
Operand 1 base address status	10	0 - base address in storage 1 - base address in register
	11	0 - do not retain base address in register 1 - retain base address in register
Operand 1 status	12	0 - generate store into operand 1 1 - do not generate store into operand 1
	13	- not used

Branch Operator (B)

```
┌──────────────────────────────────────────┐
│Chain                             (1 word)│
├──────────────────────────────────────────┤
│Branch operator                   (1 word)│
├──────────────────────────────────────────┤
│P1                                (1 word)│
├──────────────────────────────────────────┤
│                                  (1 word)│
├──────────────────────────────────────────┤
│                                  (1 word)│
├───────────┬──────────┬───────────┬─┬─────┤
│           │ Status   │           │ │     │
├───────────┴┬────┬────┴┬────┬────┬┴─┴┬────┤
│            │    │     │    │    │    │    │
└────────────┴────┴─────┴────┴────┴────┴────┘
```

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)

```
┌──────────────────────────────────────────┐
│Chain                             (1 word)│
├──────────────────────────────────────────┤
│Logical branch operator           (1 word)│
├──────────────────────────────────────────┤
│P1                                (1 word)│
├──────────────────────────────────────────┤
│P2                                (1 word)│
├──────────────────────────────────────────┤
│                                  (1 word)│
├───────────┬──────────┬───────────┬─┬─────┤
│           │ Status   │           │ │Mode │
├───────────┴┬────┬────┴┬────┬────┬┴─┴┬────┤
│            │    │     │ R2 │ B2 │    │    │
└────────────┴────┴─────┴────┴────┴────┴────┘
```

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)

```
┌──────────────────────────────────────────┐
│Chain                             (1 word)│
├──────────────────────────────────────────┤
│Binary operator                   (1 word)│
├──────────────────────────────────────────┤
│P1                                (1 word)│
├──────────────────────────────────────────┤
│P2                                (1 word)│
├──────────────────────────────────────────┤
│P3                                (1 word)│
├───────────┬──────────┬───────────┬─┬─────┤
│           │ Status   │           │ │Mode │
├───────────┴┬────┬────┴┬────┬────┬┴─┴┬────┤
│            │ R1 │ B1  │ R2 │ B2 │ R3 │ B3 │
└────────────┴────┴─────┴────┴────┴────┴────┘
```

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

```
┌──────────────────────────────────────────┐
│Chain                             (1 word)│
├──────────────────────────────────────────┤
│Test and set operator             (1 word)│
├──────────────────────────────────────────┤
│P1                                (1 word)│
├──────────────────────────────────────────┤
│P2                                (1 word)│
├──────────────────────────────────────────┤
│P3                                (1 word)│
├──────────┬─────────┬──────────┬─┬────────┤
│          │ Status  │          │ │  Mode  │
├──────────┴┬────┬───┴┬────┬────┴┬┴──┬─────┤
│           │ R1 │ B1 │ R2 │ B2 │ R3 │ B3 │
└───────────┴────┴────┴────┴────┴────┴─────┘
```

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)

```
┌──────────────────────────────────────────┐
│Chain                             (1 word)│
├──────────────────────────────────────────┤
│Function Operator                 (1 word)│
├──────────────────────────────────────────┤
│P1                                (1 word)│
├──────────────────────────────────────────┤
│P2                                (1 word)│
├──────────────────────────────────────────┤
│P3                                (1 word)│
├──────────┬─────────┬──────────┬─┬────────┤
│          │ Status  │          │ │  Mode  │
├──────────┴┬────┬───┴┬────┬────┴┬┴──┬─────┤
│           │ R1 │ B1 │ R2 │ B2 │ R3 │ B3 │
└───────────┴────┴────┴────┴────┴────┴─────┘
```

Testing a Byte Logical Variable (LDB)

```
┌──────────────────────────────────────────┐
│Chain                             (1 word)│
├──────────────────────────────────────────┤
│LDB operator                      (1 word)│
├──────────────────────────────────────────┤
│P1                                (1 word)│
├──────────────────────────────────────────┤
│P2                                (1 word)│
├──────────────────────────────────────────┤
│                                  (1 word)│
├──────────┬─────────┬──────────┬─┬────────┤
│          │ Status  │          │ │  Mode  │
├──────────┴┬────┬───┴┬────┬────┴┬┴──┬─────┤
│           │ R1 │    │ R2 │    │ 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

```
┌──────────────────────────────────┐
│ Chain                   (1 word) │
├──────────────────────────────────┤
│ Add operator            (1 word) │
├──────────────────────────────────┤
│ P1                      (1 word) │
├──────────────────────────────────┤
│ P2                      (1 word) │ Text
├──────────────────────────────────┤ Entry 1
│ P3                      (1 word) │
├──┬─────────┬─────────────┬─┬─────┤
│  │ Status  │             │ │     │
├──┴─┬────┬──┴─┬────┬────┬─┴─┴──┬──┤
│    │    │    │ R2 │    │ R3 │ B3 │
└────┴────┴────┴────┴────┴────┴────┘
```

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.

```
┌──────────────────────────────────┐
│ Chain                   (1 word) │
├──────────────────────────────────┤
│ Branch operator         (1 word) │
├──────────────────────────────────┤
│ P1                      (1 word) │
├──────────────────────────────────┤
│ P2                      (1 word) │ Text
├──────────────────────────────────┤ Entry 2
│ P3                      (1 word) │
├──┬─────────┬─────────────┬─┬─────┤
│  │ Status  │             │ │     │
├──┴─┬────┬──┴─┬────┬────┬─┴─┴──┬──┤
│    │    │    │ R2 │    │ R3 │ B3 │
└────┴────┴────┴────┴────┴────┴────┘
```

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

```
┌──────────────────────────────────────────┐
│Chain                            (1 word) │
├──────────────────────────────────────────┤
│Computed GO TO operator          (1 word) │
├──────────────────────────────────────────┤
│P1                               (1 word) │
├──────────────────────────────────────────┤
│P2                               (1 word) │
├──────────────────────────────────────────┤
│P3                               (1 word) │
├───────────┬─────────┬───────────┬─┬──────┤
│           │ Status  │           │ │      │
├───────────┴┬────┬───┴┬────┬────┬┴─┴─┬────┤
│            │    │    │    │ B2 │ R3 │ B3 │
└────────────┴────┴────┴────┴────┴────┴────┘
```

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)

```
┌────────────────────────────────────────┐
│Chain                           (1 word)│
├────────────────────────────────────────┤
│Branch operator                 (1 word)│
├────────────────────────────────────────┤
│P1                              (1 word)│
├────────────────────────────────────────┤
│P2                              (1 word)│
├────────────────────────────────────────┤
│P3                              (1 word)│
├──────────┬─────────┬──────────┬─┬──────┤
│          │ Status  │          │ │ Mode │
├──────────┴┬────┬───┴┬────┬────┼─┴─┬────┤
│           │    │    │ R2 │ B2 │ R3│ B3 │
└───────────┴────┴────┴────┴────┴───┴────┘
```

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)

```
┌────────────────────────────────────────┐
│Chain                           (1 word)│
├────────────────────────────────────────┤
│Binary shift operator           (1 word)│
├────────────────────────────────────────┤
│P1                              (1 word)│
├────────────────────────────────────────┤
│P2                              (1 word)│
├────────────────────────────────────────┤
│Shift quantity                  (1 word)│
├──────────┬─────────┬──────────┬─┬──────┤
│          │ Status  │          │ │ Mode │
├──────────┴┬────┬───┴┬────┬────┼─┴─┬────┤
│           │    │    │ R2 │ B2 │ R3│ B3 │
└───────────┴────┴────┴────┴────┴───┴────┘
```

Load Address Operator (LA)

```
┌────────────────────────────────────────┐
│Chain                           (1 word)│
├────────────────────────────────────────┤
│Load address operator           (1 word)│
├────────────────────────────────────────┤
│P1                              (1 word)│
├────────────────────────────────────────┤
│P2                              (1 word)│
├────────────────────────────────────────┤
│P3                              (1 word)│
├──────────┬─────────┬──────────┬─┬──────┤
│          │ Status  │          │S│ Mode │
├──────────┴┬────┬───┴┬────┬────┼─┴─┬────┤
│           │ R1 │    │ R2 │    │ R3│ B3 │
├───────────┴────┴────┴────┴────┴───┴────┤
│Displacement                    (1 word)│
└────────────────────────────────────────┘
```

Note: The purpose of the load address operator is to store an address of an element of an array in a parameter list. The P1 field defines the parameter list.

Subscript Text Entry - Case 1

```
┌─────────────────────────────────────────┐
│Chain                            (1 word)│
├─────────────────────────────────────────┤
│Subscript operator               (1 word)│
├─────────────────────────────────────────┤
│P1                               (1 word)│
├─────────────────────────────────────────┤
│P2                               (1 word)│
├─────────────────────────────────────────┤
│P3                               (1 word)│
├───────────┬──────────┬──────────┬─┬─────┤
│           │  Status  │          │S│ Mode│
├───────────┴┬────┬────┴┬────┬────┴─┴┬────┤
│            │ R1 │ B1  │ R2 │ B2 │ R3 │ B3 │
├────────────┴────┴─────┴────┴────┴────┴────┤
│Displacement                     (1 word)│
└─────────────────────────────────────────┘
```

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.

Subscript Text Entry - Case 2

```
┌─────────────────────────────────────────┐
│Chain                            (1 word)│
├─────────────────────────────────────────┤
│Subscript operator               (1 word)│
├─────────────────────────────────────────┤
│                                 (1 word)│
├─────────────────────────────────────────┤
│P2                               (1 word)│
├─────────────────────────────────────────┤
│P3                               (1 word)│
├───────────┬──────────┬──────────┬─┬─────┤
│           │  Status  │          │S│ Mode│
├───────────┴┬────┬────┴┬────┬────┴─┴┬────┤
│            │    │     │    │ B2 │ R3 │ B3 │
├────────────┴────┴─────┴────┴────┴────┴────┤
│Displacement                     (1 word)│
└─────────────────────────────────────────┘
```

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.

In-line routines (DABS, ABS, IABS, IDINT, INT, HFIX, DFLOAT, FLOAT, DBLE)

```
┌─────────────────────────────────────────┐
│Chain                            (1 word)│
├─────────────────────────────────────────┤
│Operator                         (1 word)│
├─────────────────────────────────────────┤
│P1                               (1 word)│
├─────────────────────────────────────────┤
│P2                               (1 word)│
├─────────────────────────────────────────┤
│                                 (1 word)│
├───────────┬──────────┬──────────┬─┬─────┤
│           │  Status  │          │ │ Mode│
├───────────┴┬────┬────┴┬────┬────┴─┴┬────┤
│            │ R1 │ B1  │ R2 │ B2 │    │    │
└────────────┴────┴─────┴────┴────┴────┴────┘
```

EXT, and LIBF Operators

```
┌──────────────────────────────────────────┐
│Chain                             (1 word)│
├──────────────────────────────────────────┤
│Operator                          (1 word)│
├──────────────────────────────────────────┤
│P1                                (1 word)│
├──────────────────────────────────────────┤
│P2                                (1 word)│
├──────────────────────────────────────────┤
│P3                                (1 word)│
├──────────┬──────────┬──────────┬─┬───────┤
│          │ Status   │          │ │       │
├──────────┴┬────┬────┴┬────┬────┴┬┴──┬────┤
│           │ R1 │ B1  │    │     │   │    │
└───────────┴────┴─────┴────┴─────┴───┴────┘
```

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

```
┌──────────────────────────────────────────┐
│Chain                             (1 word)│
├──────────────────────────────────────────┤
│Argument operator                 (1 word)│
├──────────────────────────────────────────┤
│P1                                (1 word)│
├──────────────────────────────────────────┤
│P2                                (1 word)│
├──────────────────────────────────────────┤
│P3 (for complex)                  (1 word)│
├──────────┬──────────┬──────────┬─┬───────┤
│          │          │          │ │       │
├──────────┴┬────┬────┴┬────┬────┴┬┴──┬────┤
│           │    │     │    │     │   │    │
└───────────┴────┴─────┴────┴─────┴───┴────┘
```

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

```
┌──────────────────────────────────────────┐
│Chain                             (1 word)│
├──────────────────────────────────────────┤
│Argument operator                 (1 word)│
├──────────────────────────────────────────┤
│P1                                (1 word)│
├──────────────────────────────────────────┤
│                                  (1 word)│
├──────────────────────────────────────────┤
│                                  (1 word)│
├──────────┬──────────┬──────────┬─┬───────┤
│          │ Status   │          │ │       │
├──────────┴┬────┬────┴┬────┬────┴┬┴──┬────┤
│           │ 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)

```
┌────────────────────────────────────────┐
|Chain                          (1 word)|
├────────────────────────────────────────┤
|Assigned GO TO operator        (1 word)|
├────────────────────────────────────────┤
|                               (1 word)|
├────────────────────────────────────────┤
|P2                             (1 word)|
├────────────────────────────────────────┤
|                               (1 word)|
├───────────┬──────────┬───────┬─┬──────┤
|           | Status   |       | |      |
├───────────┴─┬────┬───┴┬────┬─┴─┴┬─────┤
|             |    |    | R2 | B2 |     |     |
└─────────────┴────┴────┴────┴────┴─────┴─────┘
```

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

READ/WRITE Operators for I/O lists

READ

```
┌────────────────────────────────────────┐
|Chain                          (1 word)|
├────────────────────────────────────────┤
|READ operator                  (1 word)|
├────────────────────────────────────────┤
|P1                             (1 word)|
├────────────────────────────────────────┤
|                               (1 word)|
├────────────────────────────────────────┤
|P3                             (1 word)|
├───────────┬──────────┬───────┬─┬──────┤
|           | Status   |       | |      |
├───────────┴─┬────┬───┴┬────┬─┴─┴┬─────┤
|             | R1 | B1 |    |    |     |     |
└─────────────┴────┴────┴────┴────┴─────┴─────┘
```

P1: The P1 field contains a pointer to the I/O list for the READ statement.

Note: If the P3 field contains a zero, an entire array is being read. This causes a different instruction sequence to be generated.

WRITE

```
┌────────────────────────────────────────┐
|Chain                          (1 word)|
├────────────────────────────────────────┤
|WRITE operator                 (1 word)|
├────────────────────────────────────────┤
|                               (1 word)|
├────────────────────────────────────────┤
|P2                             (1 word)|
├────────────────────────────────────────┤
|P3                             (1 word)|
├───────────┬──────────┬───────┬─┬──────┤
|           | Status   |       | |      |
├───────────┴─┬────┬───┴┬────┬─┴─┴┬─────┤
|             | R1 | B1 |    |    |     |     |
└─────────────┴────┴────┴────┴────┴─────┴─────┘
```

P2: The P2 field contains a pointer to the I/O list for the WRITE statement.

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)

```
┌──────────────────────────────────────────┐
│Chain                             (1 word)│
├──────────────────────────────────────────┤
│Logical Branch Operator           (1 word)│
├──────────────────────────────────────────┤
│P1                                (1 word)│
├──────────────────────────────────────────┤
│P2                                (1 word)│
├──────────────────────────────────────────┤
│P3                                (1 word)│
├──────────┬─────────┬───────────┬─┬───────┤
│          │ Status  │           │ │ Mode  │
├──────────┴┬────┬───┴┬────┬────┬┴─┴──┬────┤
│           │ R1 │    │    │ B2 │     │    │
└───────────┴────┴────┴────┴────┴─────┴────┘
```

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

```
┌──────────────────────────────────────────┐
│Chain                             (1 word)│
├──────────────────────────────────────────┤
│LBIT Operator                     (1 word)│
├──────────────────────────────────────────┤
│P1                                (1 word)│
├──────────────────────────────────────────┤
│P2                                (1 word)│
├──────────────────────────────────────────┤
│P3                                (1 word)│
├──────────┬─────────┬───────────┬─┬───────┤
│          │ Status  │           │ │ Mode  │
├──────────┴┬────┬───┴┬────┬────┬┴─┴──┬────┤
│           │    │    │    │ B2 │     │    │
└───────────┴────┴────┴────┴────┴─────┴────┘
```

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.

APPENDIX C: ARRAYS

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.

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. (In some cases, 0's rather than X's appear for base register loads and the subject store instruction.)

MINUS: Used for All Subtract Operations

Index	Skeleton Instructions	Status
		0000000011111111
		0000111100001111
		0011001100110011
		0101010101010101
1	L B2,D(0,BD)	XXXXXXXX00000000
2	LH R2,D(0,B2)	0000111100000000
3	LH R1,D(X,B2)	1100000000000000
4	L B3,D(0,BD)	XX00XX00XX00XX00
5	LCR R3,R3	0010001000000010
6	LR R1,R2	0000110100001101
7	LH R3,D(0,B3)	0100010001000100
8	LCR R1,R3	0001000000000000
9	SH R1,D(X,B3)	1000100010001000
10	SR R1,R3	0100010101110101
11	AH R3,D(X,B2)	0010000000000000
12	AH R1,D(X,B2)	0001000000000000
13	AR R3,R2	0000001000000010
14	L B1,D(0,BD)	XXXXXXXXXXXXXXXX
15	STH R1,D(0,B1)	XXXXXXXXXXXXXXXX

---

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

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

Index	Skeleton Instructions	INT, IFIX, HFIX Status	IDINT Status
		0011	0011
		0101	0101
1	SDR 0,0	1111	0000
2	L B2,D(0,BD)	XX00	XX00
3	LD R2,D(0,B2)	0100	0100
4	LD 0,D(0,B2)	1000	1000
5	LDR 0,R2	0111	0111
6	AW 0,60(0,12)	1111	1111
7	STD 0,64(0,13)	1111	1111
8	L R1,68(0,13)	1111	1111
9	BALR 15,0	1111	1111
10	BC 10,6(0,15)	1111	1111
11	LNR R1,R1	1111	1111
12	L B1,D(0,BD)	XXXX	XXXX
13	STH R1,D(0,B1)	XXXX	XXXX

ABSGEN: 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

MOD24: 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

MXMNGN: 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.

SHFTRL: 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

SIGNGN: 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

DBLGEN: 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

CMPLGN: 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

ADMDGN: 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 ADMDGN
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.

LGLNOT: 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

DIMGEN: 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

BTBF: 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 depending on the operation.		

LDADDR: 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

LDBGEN: 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

SUBGEN: 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

UNRGEN: 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

BRCOMB: 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

BRCOMP: Used for All Assigned GO TO Operations

Index	Skeleton Instructions	Status
		0000000011111111
		0000111100001111
		0011001100110011
		0101010101010101
1	L B2,D(0,BD)	0000000000000000
2	L R2,D(0,B2)	1111111100000000
3	BCR 15,R2	1111111111111111

STRGEN: 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

INTMPY: 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

DIVGEN: 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.		

DIVGEN: 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.		

TSTSET: 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

LOGCL: 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

PLSGEN: 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.

FLTGEN: 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

NDORGN: 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

SHFT2: 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

BRLGL: Used for Text Entries Whose Operator is a Relational Operator Operating on Two Non-Zero 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

BRLGL: 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

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

Index	Skeleton Instructions	BBT,BBF		LBIT	
		simple variable	subscripted variable	simple variable	subscripted variable
1	L B2,D(0,BD)	X	X	X	X
2	LA 15,D+N/8(X,B2)	0	1	0	1
3	TM M,D+N/8(B2)	1	0	1	0
4	TM M,0(15)	0	1	0	1
5	TM M,D+N/8(R2)	0	0	0	0
6	L 15,P1	1	1	0	0
7	BCR MM,15	1	1	0	0
8	BALR 15,0	0	0	1	1
9	LA R1,1(0,0)	0	0	1	1
10	BC 1,10(0,15)	0	0	1	1
11	SR R1,R1	0	0	1	1
12	L B1,D(0,BD)	0	0	X	X
13	ST R1,D(0,B1)	0	0	X	X

N = The bit to be loaded or tested.

M = MSKTBL(MOD(N,8)+1). MSKTBL is an array of masks used by LBITTF.

MM = 1 FOR BBT.

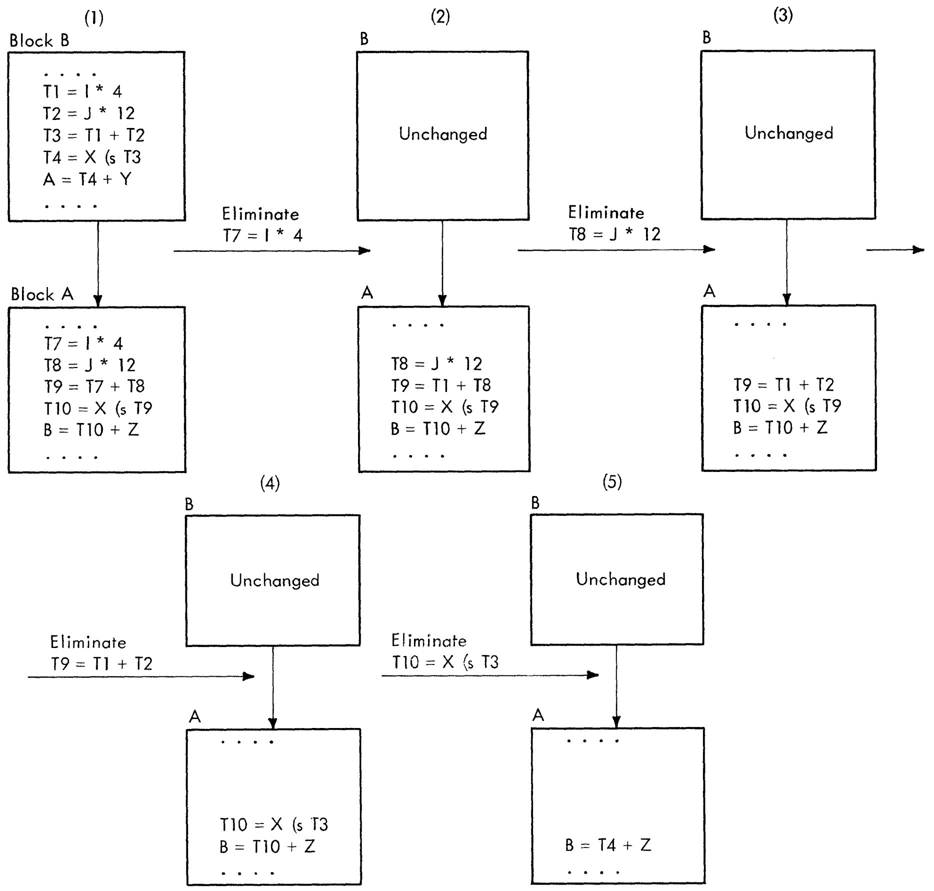
MM = 8 FOR BBF.

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

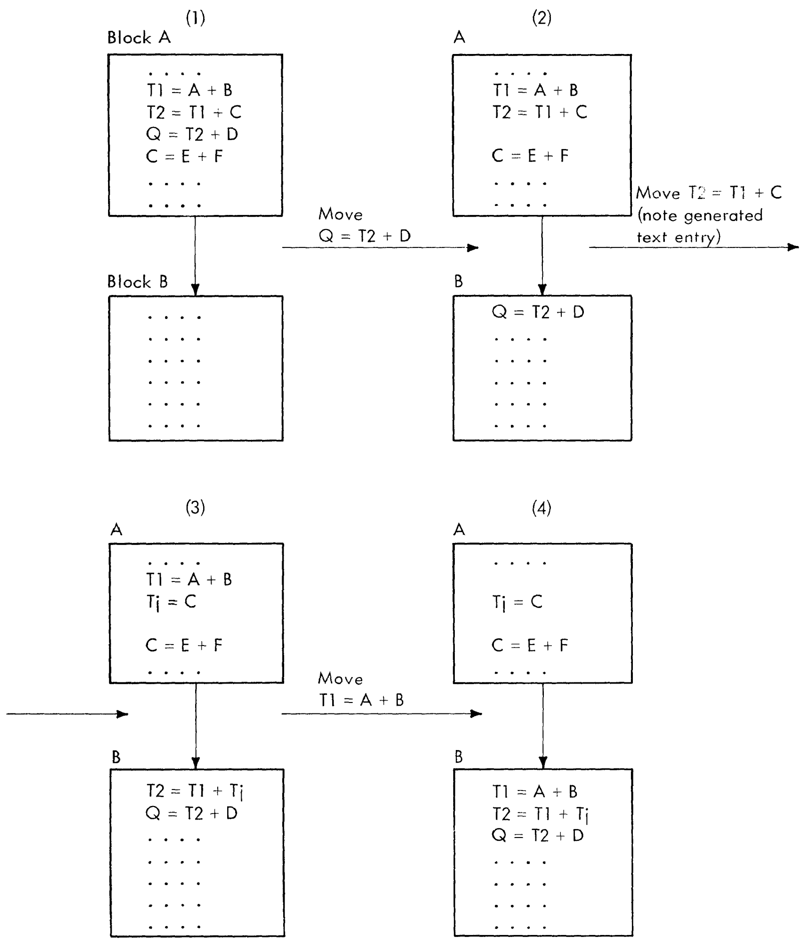
This example illustrates the concept of common expression elimination. The text entries in block A are to undergo common expression elimination. Block B is a back dominator of block A. Block B contains text entries that are common to those in block A.



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

Example 2: Forward Movement

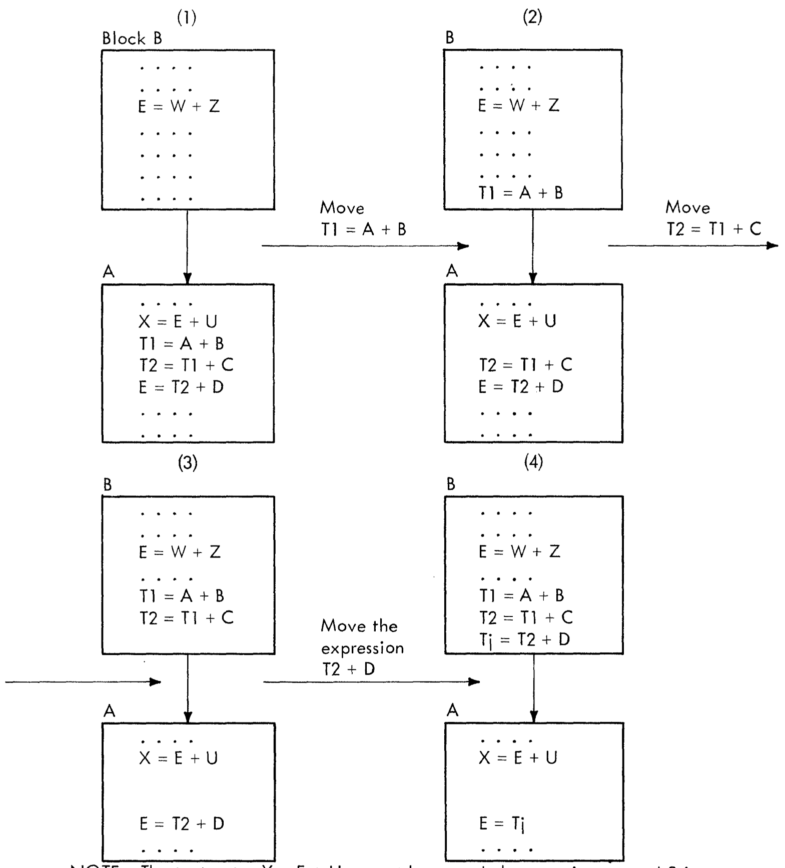
This example illustrates both methods of forward movement. Block A, containing the text entries to be moved, is a back dominator of the forward target of the loop, which is block B.



NOTE: The text entry C = E + F cannot be moved, because operand 1 (C) is used elsewhere in the loop

Example 3: Backward Movement

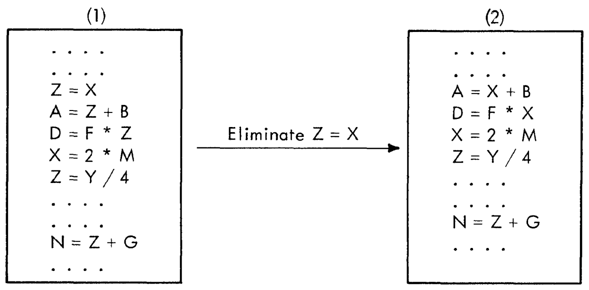
This example illustrates both methods of backward movement. The text entries in block A are to undergo backward movement. Block B is the back target of the loop containing block A.



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 that the characteristics of the operands of the simple store correspond to the last combination of characteristics stated in Table 4.



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: Constant Expression Reordering

The following text and figures describe and illustrate the concept of constant expression reordering. The text entries to be operated upon and the type 5 and type 6 tables are shown. Candidate pairs are indicated by arrows. Note that reordering involving type6-type5 candidate pairs causes the text entries to switch roles (i.e., switch tables). In this example text entries to compute the result of the interacting constants do not appear in the back target, because, in all cases, both constants are absolute.

The initial text entries and table entries are:

TYPE 5	TYPE 6	TEXT
T4=T3*4.0	T3=T2+8.0	T1=N-3.0
		T2=T1*5.0
T2=T1*5.0	T1=N-3.0	T3=T2+8.0
		T4=T3*4.0
		A=X(T4

The type 5 text entries do not constitute a candidate pair because they lack a common temporary. However, three type 6-type 5 candidate pairs exist and undergo constant expression reordering.

TYPE 5	TYPE 6	TEXT
T4=T3*4.0 ←	T3=T2+8.0	T1=N-3.0
		T2=T1*5.0
T2=T1*5.0	T1=N-3.0	T3=T2*4.0
		T4=T3+32.0
		A=X(T4

TYPE 5	TYPE 6	TEXT
T3=T2*4.0	T4=T3+32.0	T1=N*5.0
		T2=T1-15.0
T2=T1*5.0 ←	T1=N-3.0	T3=T2*4
		T4=T3+32.0
		A=X(T4

TYPE 5	TYPE 6	TEXT
T3=T2*4.0	T4=T3+32.0	T1=N*5.0
	↖	T2=T1*4.0
T1=N*5.0	T2=T1-15.0	T3=T2-60.0
		T4=T3+32.0
		A=X(T4

The remaining type 5 text entries constitute a candidate pair and undergo constant expression reordering.

TYPE 5	TYPE 6	TEXT
T2=T1*4.0	T4=T3+32.0	T2=N*20.0
↑		T3=T2-60.0
T1=N*5.0	T3=T2-60.0	T4=T3+32.0
		A=X(T4

The type 6 text entries are candidates and are processed.

TYPE 5	TYPE 6	TEXT
T2=N*20.0	T4=T3+32.0	T2=N*20.0
	↑	T4=T2+(-28.0)
	T3=T2-60.0	A=X(T4

The type 6-type5 pair that remains does not constitute a candidate pair, because the text entry in which the common temporary is defined does not have an additive operator.

TYPE 5	TYPE 6
T2=N*20.0	T4=T2+(-28.0)

Because operand 1 of the remaining type 6 text entry is used as a subscript, the type 6 text entry can be eliminated by replacing the subscript with the used temporary of that text entry and incorporating the additive constant of that text entry into the displacement of the subscripted variable. The resultant text appears as:

```
T2=N*20.0
A=X( T2
```

Example 5: 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:

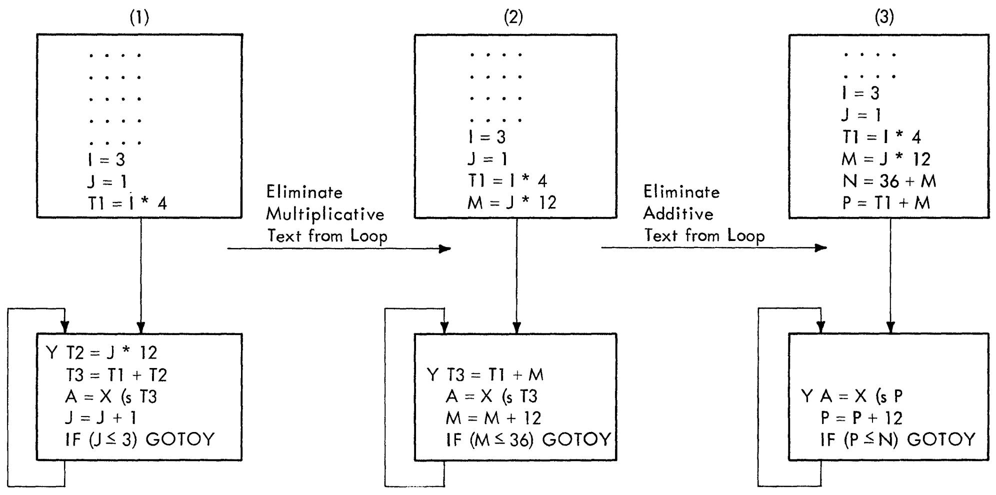
```
    I=3
    DO 10 J=1,3
    A=X(I,J)
10  CONTINUE
```

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 (T3	Stores X(I,J) into A	The phase 10 text entry forced and converted to phase 15 text after the index value for the subscripted variable has been established.
	J = J + 1	Increments DO index.	Generated by phase 10 and converted to phase 15 text representation.
	IF(J≤3)GOTO Y	Tests DO index against its maximum and controls branching.	Generated by phase 10 and converted to phase 15 text representation.

Note: The statement number Y is generated by phase 10. Also, it is assumed that the array X is of the form X(3,3) and that its elements are real (length 4).

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



IHCFCOMH, a member of the FORTRAN system library (SYS1.FORTLIB), performs object-time implementation of the following FORTRAN source statements:

- READ and WRITE.
- BACKSPACE, REWIND, and END FILE (device manipulation).
- STOP and PAUSE (write to operator).

In addition, IHCFCOMH processes object-time errors detected by the various FORTRAN library subprograms, processes arithmetic-type program interruptions, and terminates load module execution. The load module is produced by the linkage editor and contains:

- The object module produced by the compiler.
- IHCFCOMH.
- IHCFIOSH.
- Required subprograms.

All linkages from the object module (produced by the compiler) to IHCFCOMH are via compiler-generated calling sequences. Each time one of the above mentioned source statements is encountered during the compilation, an appropriate calling sequence to IHCFCOMH is generated and included as part of the object module. At object time, these calls are executed, and control is passed to IHCFCOMH to perform the specified operation.

The routines of IHCFCOMH are divided into the following categories:

- READ/WRITE routines.
- Device manipulation routines.
- Write-to-operator routines.
- Utility routines.
- Conversion routines.

The overall logic of IHCFCOMH is illustrated in Chart 24.

CONSIDERATION: IHCFCOMH, itself, does not perform the actual initialization of data sets, reading/writing of data sets, or device manipulation. It submits requests for such operations to the FORTRAN Input/Output System (IHCFIOSH), which is discussed in Appendix F. IHCFIOSH, in turn, interprets the requests and submits them to BSAM (Basic Sequential Access Method) for 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 READ/WRITE statements not using NAMELIST, IHCFCOMH consists of the opening, I/O list, and closing sections. 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.

OPENING SECTION: The compiler generates a calling sequence to the opening section of IHCFCOMH when it detects a READ/WRITE statement that does not use a NAMELIST.

The opening section determines the nature of the operation (READ or WRITE and whether or not a format is required) and initializes the data set for reading or writing. Subsequent opening section processing is dependent upon the nature of the operation.

READ Requiring a Format: If the operation is that of READ requiring a format, the opening section reads a record, containing data to be input to the list items, into an I/O buffer. The location and size of the record are saved, a pointer to the I/O buffer is initialized to the first location in that record, and the address of the FORMAT statement associated with the READ is saved. (The address of the FORMAT statement is passed as an argument to the opening section.) If the FORMAT statement is of the variable type, control is passed to a portion of IHCFCOMH that translates variable FORMAT statements to acceptable form. After the translation, the scan of the FORMAT statement is then initiated. If the FORMAT statement is not variable, it is in acceptable form and the scan of the statement is immediately initiated. The first format code (either control or conversion type) of the FORMAT statement 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 control routine associated with the format code under consideration to perform the indicated operation (see Table 28). Control then passes to the scan portion, which obtains the next format code. The above operation is repeated for all control type codes until either the end of the FORMAT statement or the first conversion type code is encountered.

A conversion type code (e.g., an I format code) requires an I/O list item. Upon first encounter of a conversion type 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 of the calling routine within the load module. The calling routine obtains the list item associated with the conversion code and calls the I/O list section of IHCFCOMH.

<u>WRITE Requiring a Format:</u> If the operation is that of WRITE requiring a format, the opening section proceeds in a manner similar to its processing of a READ requiring a

Table 28. Processing of Format Codes

FORMAT Code	Description	Type	Corresponding Action Upon Code
	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 object program to return control to subroutine FIOLF or FIOAF. 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 current position in the I/O buffer, the scale factor, and the values of w and d.
)	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 using subroutine IHCFIOSH.
	end of statement	control	If no I/O list items remain to be transmitted, return control to the object program to link to subroutine FENDF, the closing section; if list items remain, input or output one record using subroutine IHCFIOSH. Repeat format codes from last first level left parenthesis.

format. However, instead of reading a record, the opening section obtains and initializes an I/O buffer for output.

READ Not Requiring a Format: If the operation is that of READ not requiring a format, the opening section of IHCFCOMH reads a record into an I/O buffer. This section saves the location and size of the record, initializes the buffer pointer and returns control to the next sequential instruction of the calling routine within the load module. The calling routine obtains a list item and calls the I/O list section of IHCFCOMH.

WRITE Not Requiring a Format: If the operation is that of WRITE not requiring a format, the opening section of IHCFCOMH proceeds in a manner similar to its processing of a READ not requiring a format. However, instead of reading a record, the opening section obtains and initializes an I/O buffer for output.

I/O LIST SECTION: The compiler generates a calling sequence to the I/O list section of IHCFCOMH when it encounters an I/O list item associated with a READ/WRITE statement that does not use a NAMELIST.

The I/O list section performs the actual input of data to the list item if a READ statement is being implemented, and the actual output of data from the list item if a WRITE statement is being implemented.

READ/WRITE Requiring a Format: In processing a list item for any READ or WRITE requiring a format, the I/O list section passes control to the conversion routine that puts the list item in a format according to its associated conversion type format code. (The appropriate conversion routine is determined by the scan portion of IHCFCOMH. Its selection is a function of the format code being processed by the scan portion. The address of the conversion routine is made available to the I/O list section.) For input, the conversion routine obtains data from the I/O buffer and converts the data to the form dictated by the format code. The converted data is then moved into the list item. For output, the conversion routine obtains the data in the list item, converts it to the form dictated by the format code, and places the converted result in the I/O buffer.

After the conversion routine has processed the list item, the I/O list section determines if the format code applied to the list item just processed is to be repeated for the next list item. It looks at the field count (if any) associated with the format code. (The field count indicates the number of times a particular conversion type format code is to be repeated for successive list items.) If the format code is to be repeated and if the item just processed was a variable, control is returned to the calling routine within the load module. The calling routine obtains the next list item and again links to the I/O list section. The conversion routine that processed the previous list item is then given control. This action applies the same format code to the new list item.

If the format code is to be repeated and the list item just processed was an array element, the next element of the array is obtained. The format code is repeated for this element. There is no return to the calling routine of the load module until all of the array elements have been satisfied. If the format 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 format codes is to repeated, the FORMAT statement pointer is adjusted to the first code in the group. The codes of the group are then repeated. If a group is not to be repeated, the next format code is obtained. For a control type code, control is passed to its associated control routine. For a conversion type code, control is returned to the calling routine within the load module, which obtains the list item associated with the conversion code. The calling routine again links to the I/O list section to process the list item.

READ/WRITE Not Requiring a Format: In processing list items for a READ or WRITE statement not requiring a format, the I/O list section determines the size of the list item (i.e., the number of bytes reserved for the list item). 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 I/O buffer to the list item on input, and reversed on output. Control is then returned to the calling routine within the load module to obtain the next list item.

CLOSING SECTION: The compiler generates a linkage to the closing section of IHCFCOMH after it has processed all list items associated with a READ/WRITE statement that does not use NAMELIST. The closing section terminates input/output operations.

READ/WRITE Requiring a Format: If a READ operation is being implemented, the closing section simply returns control to the calling routine within the load module. If the operation is a WRITE the last record is written and control is returned to the calling routine.

READ/WRITE Not Requiring a Format: If a READ is being implemented, successive records are read until the record that indicates the end of the logical record is recognized. (A FORTRAN logical record consists of the total number of records necessary to contain all I/O list items within a single WRITE statement.) Control is then returned to the calling routine within the load module. If the operation is WRITE, the record count (i.e., the number of records in the logical record) is placed into the control word field of the last record, the last record is written, and control is returned to the calling routine.

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.

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/O 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.

DEVICE MANIPULATION ROUTINES

The device manipulation routines of IHCFCOMH implement the BACKSPACE, REWIND, and END FILE source statements. These

---

[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).

routines receive control from object-time execution of calling sequences that are generated by the compiler when those statement types are encountered.

The implementation of REWIND and END FILE statements is straight-forward. The device manipulation routines submit the appropriate control request to IHCFIOSH, the I/O interface module. Control is then 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 if the record backspaced over is an element of a data set that does not require a format. If not, control is returned to the calling routine. If the record is an element of such a data set, that record is read into an I/O buffer and the record count is obtained from its control word. Backspace control requests, equal to the record count, are then issued and control is returned 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 the object-time execution of calling sequences that are generated by the compiler upon recognition of the statement types.

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.
- Terminate load module execution.
- Aid IHCFIOSH in processing I/O errors and end-of-data set.

ERROR MESSAGES: The error message processing routine receives control from various FORTRAN library subprograms when they detect object-time errors.

Error message processing consists of initializing the data set upon which the message is to be written, and writing the message. Control is then passed to the load module termination routine of IHCFOMH.

ARITHMETIC INTERRUPTIONS: The arithmetic interruption routine of IHCFCOMH initially receives control from the object-time execution of a compiler-generated calling sequence. The call is placed at the start of the executable code of a main program so that this routine is given control to set the program interruption mask. Subsequent entries into this routine are via arithmetic-type interruptions.

This routine sets the program interruption mask by means of a SPIE macro-instruction. The instruction specifies the type of arithmetic interruptions that are to cause control to be passed to the 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.

The first step taken by this routine in processing arithmetic interruptions is to determine its type. If exponential overflow or underflow has occurred, the appropriate indicators, which are referenced by subroutine OVERFL (a FORTRAN library subprogram), are set. If any type of divide check has caused the interrupt, the indicator referenced by subroutine DVCHK (also a FORTRAN library subprogram) is set.

Regardless of the type of interruption that has given control to it, the arithmetic interruption routine writes out the old program PSW for diagnostic purposes.

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

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

This routine terminates load module execution by closing all FORTRAN data sets that are open, by issuing a SPIE macro-instruction with no parameters to indicate the IHCFCOMH no longer desires to give special treatment to program interruptions and does not want maskable interruptions to occur, and by returning control to the operating system.

I/O ERRORS AND END-OF-DATA SET: The routines of IHCFCOMH, which aid IHCFIOSH in processing I/O errors and end-of-data set, receive control from IHCFIOSH when such events are encountered during reading or writing.

If an end-of-data set has been encountered, a check is made to determine if the END='address' parameter was specified in the READ/WRITE. If the parameter is present, control is returned to the address indicated in the parameter. If the parameter is not present, an error is signaled and control is passed to the error message processing routine of IHCFCOMH.

A similar procedure is followed when an I/O error has been encountered; however, in this case, it is determined if the ERR= 'address' parameter was specified in the READ/WRITE.

CONVERSION ROUTINES

The conversion routines of IHCFCOMH 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.

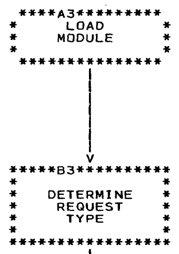
If a READ statement is being implemented, the conversion routine obtains data from the I/O buffer, converts it according to its associated conversion type format code or type, and enters the converted data into the list item. The process is reversed if a WRITE statement is being implemented.

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.

Chart 24. IHCFCOMH Logic and Utility Rtn

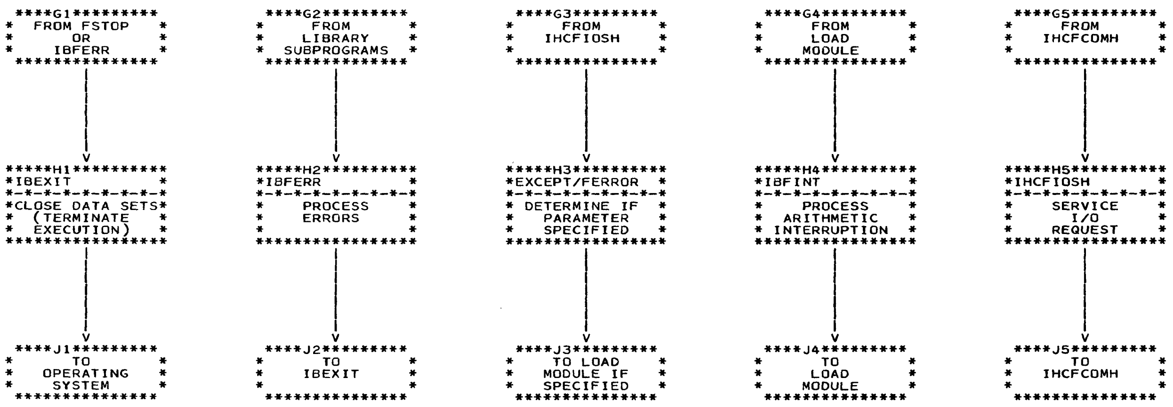
SEE TABLE 29 FOR A BRIEF DISCUSSION OF EACH ROUTINE OF IHCFCOMH.

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



REQUEST TYPE	CHART	MAJOR PROCESSING ROUTINES	SUBROUTINES CALLED
READ/WRITE RE- QUIRING A FORMAT	25A2	FRDWF,FWRWF,FIOLF, FIOAF,FENDF	IHCFIOSH,FCVII,FCVIO,FCVEI,FCVEO, FCVDI,FCVDO,FCVLI,FCVLO,FCVZI,FCVZO, FCVFI,FCVFO,FCVAI,FCVAO,FCVGI,FCVGO
READ/WRITE NOT REQUIRING A FORMAT	25F2	FRDNF,FWRNF,FIOLN, FIOAN,FENDN	IHCFIOSH
READ USING NAMELIST	26E1	FRDNL	IHCFIOSH,FCVEI,FCVDI,FCVAI, FCVLI,FCVGI,FCVCI,FCVFI,FCVII
WRITE USING NAMELIST	26E5	FWDNL	IHCFIOSH,FCVEO,FCVDO,FCVAO,FCVLO, FCVGO,FCVCO,FCVIO,FCVFO
DEVICE MANIPULATION	26B3	FBKSP,FRWND, FEOFM	IHCFIOSH
WRITE TO OPERATOR	26G3	FSTOP,FPAUS	NONE

UTILITY ROUTINES



IF PARAMETER NOT SPECIFIED, EXIT IS TO IBFERR

Chart 25. Implementation of RD/WR Srce Stmnts

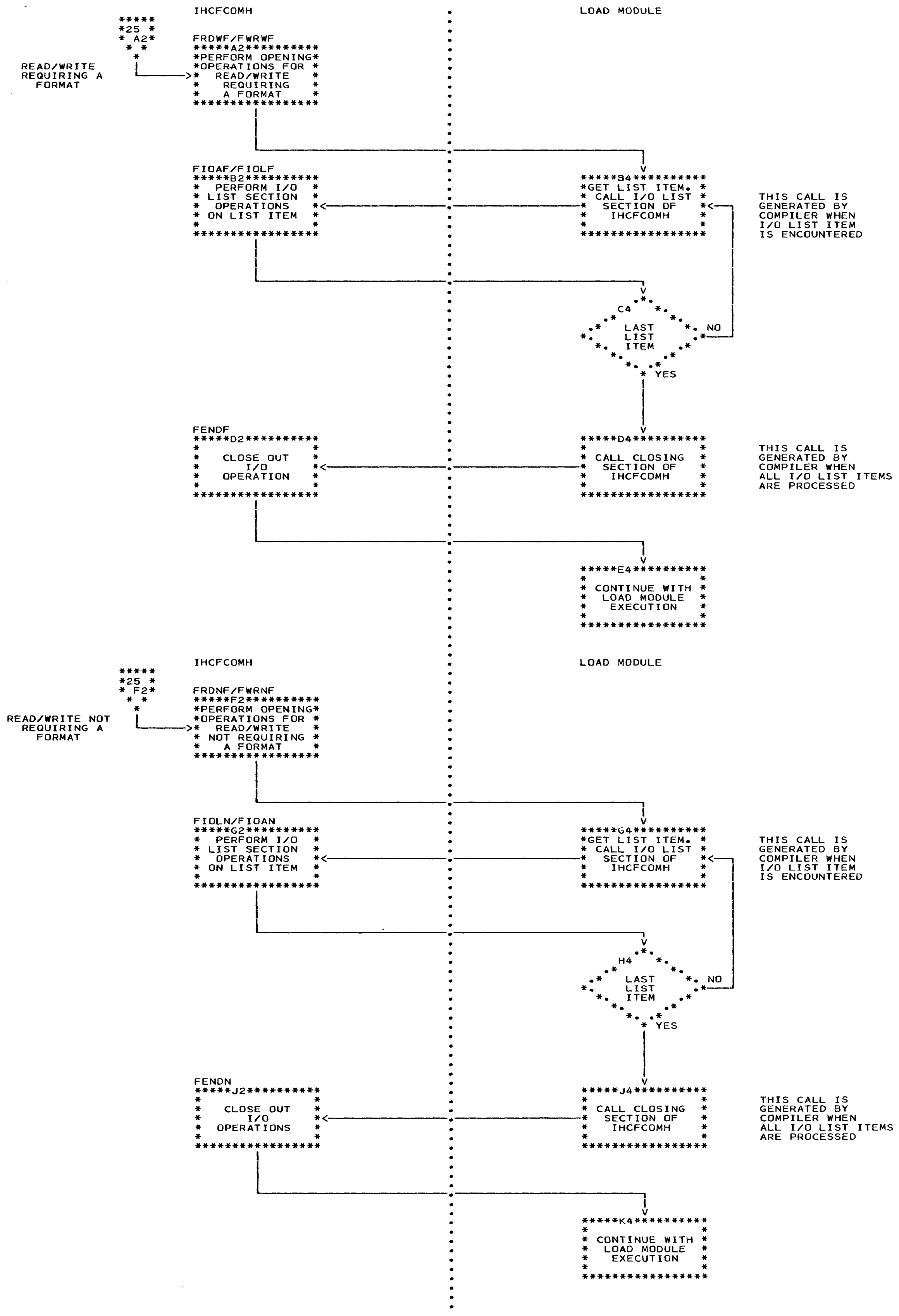

IHCFCOMH
LOAD MODULE
25 A2
READ/WRITE REQUIRING A FORMAT
FRDWF/FWRWF
A2
PERFORM OPENING OPERATIONS FOR READ/WRITE REQUIRING A FORMAT
FIOAF/FIOLF
B2
PERFORM I/O LIST SECTION OPERATIONS ON LIST ITEM
B4
GET LIST ITEM. CALL I/O LIST SECTION OF IHCFCOMH
THIS CALL IS GENERATED BY COMPILER WHEN I/O LIST ITEM IS ENCOUNTERED
C4
LAST LIST ITEM
NO
YES
FENDF
D2
CLOSE OUT I/O OPERATION
D4
CALL CLOSING SECTION OF IHCFCOMH
THIS CALL IS GENERATED BY COMPILER WHEN ALL I/O LIST ITEMS ARE PROCESSED
E4
CONTINUE WITH LOAD MODULE EXECUTION
IHCFCOMH
LOAD MODULE
25 F2
READ/WRITE NOT REQUIRING A FORMAT
FRDNF/FWRNF
F2
PERFORM OPENING OPERATIONS FOR READ/WRITE NOT REQUIRING A FORMAT
FIOLN/FIOAN
G2
PERFORM I/O LIST SECTION OPERATIONS ON LIST ITEM
G4
GET LIST ITEM. CALL I/O LIST SECTION OF IHCFCOMH
THIS CALL IS GENERATED BY COMPILER WHEN I/O LIST ITEM IS ENCOUNTERED
H4
LAST LIST ITEM
NO
YES
FENDN
J2
CLOSE OUT I/O OPERATIONS
J4
CALL CLOSING SECTION OF IHCFCOMH
THIS CALL IS GENERATED BY COMPILER WHEN ALL I/O LIST ITEMS ARE PROCESSED
K4
CONTINUE WITH LOAD MODULE EXECUTION

Chart 26. Dvce Mnpltn, WR to Oprtr, RD/WR NMLST

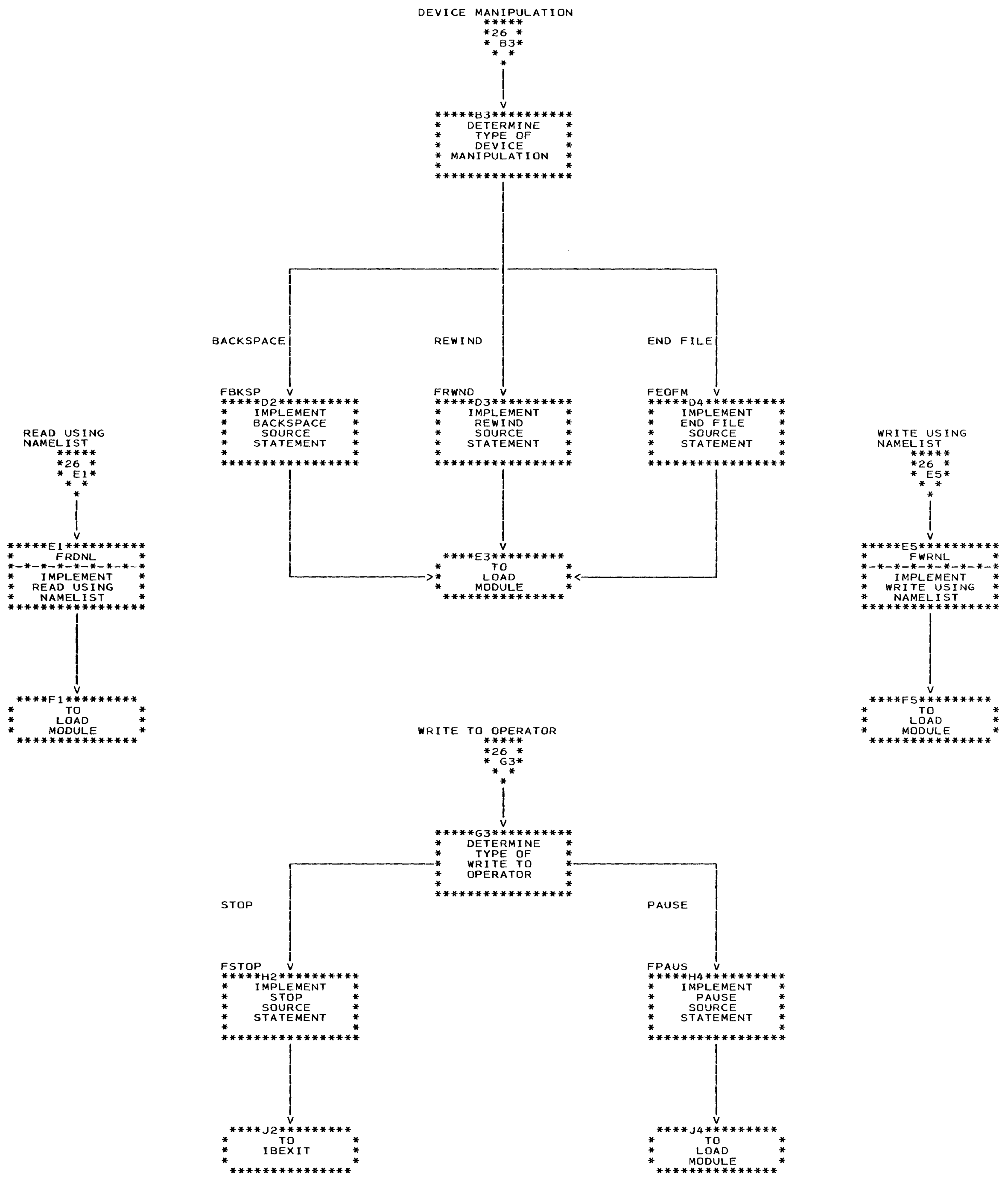

DEVICE MANIPULATION
26 B3
B3 DETERMINE TYPE OF DEVICE MANIPULATION
BACKSPACE
REWIND
END FILE
FBKSP
D2 IMPLEMENT BACKSPACE SOURCE STATEMENT
FRWND
D3 IMPLEMENT REWIND SOURCE STATEMENT
FEOFM
D4 IMPLEMENT END FILE SOURCE STATEMENT
E3 TO LOAD MODULE
READ USING NAMELIST
26 E1
E1 FRDNL IMPLEMENT READ USING NAMELIST
F1 TO LOAD MODULE
WRITE USING NAMELIST
26 E5
E5 FWRNL IMPLEMENT WRITE USING NAMELIST
F5 TO LOAD MODULE
WRITE TO OPERATOR
26 G3
G3 DETERMINE TYPE OF WRITE TO OPERATOR
STOP
PAUSE
FSTOP
H2 IMPLEMENT STOP SOURCE STATEMENT
FPAUS
H4 IMPLEMENT PAUSE SOURCE STATEMENT
J2 TO IBEXIT
J4 TO LOAD MODULE

Table 29. IHCFCOMH Subroutine Directory

Subroutine	Function
EXCEPT	Checks for presence of END= parameter, and passes control to the load module if present.
FBKSP	Implements the BACKSPACE source statement.
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.
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.
FRDNL	Processes READ statements using NAMELISTs.
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.
FWRNL	Processes WRITE statements using NAMELISTs.
FWRWF	Opening section for WRITE requiring a format.
IBEXIT	Closes all data sets and terminates execution.
IBFERR	Processes object-time errors.
IBFINT	Processes arithmetic-type program interruptions.
IHCFIOSH	Services I/O requests from IHCFCOMH.

IHCFIOSH, the object time FORTRAN Input/Output System, resides on the FORTRAN system library (SYS1.FORTLIB). Its function is to receive input/output requests for IHCFCOMH and submit them to the access method BSAM (Basic Sequential Access Method) for implementation.

BLOCKS AND TABLE

IHCFIOSH uses the following blocks and table during its processing of input/output requests:

- Unit blocks.
- Unit assignment table.

UNIT BLOCKS

IHCFIOSH generates unit blocks for the unit numbers (i.e., data set reference numbers) used in the various input/output operations. The first input/output operation using a particular unit number causes IHCFIOSH to obtain (via a GETMAIN macro-instruction) a block of storage for use as the unit block for the specified unit. Each unit block has the following format:

```
┌────────┬────────┬────────┬────────┐
│Data Set│        │        │        │
│ Type   │        │        │        │
│(ABYTE) │ BBYTE  │ CBYTE  │LIVECNT │
├────────┴────────┴────────┴────────┤
│            Buffer 1               │
├───────────────────────────────────┤  House-
│            Buffer 2               │  keeping
├───────────────────────────────────┤  Section
│          Block Pointer            │
├───────────────────────────────────┤
│          Record Offset            │
├───────────────────────────────────┤
│                                   │
│       DECB SKELETON SECTION       │
│                                   │
├───────────────────────────────────┤
│                                   │
│                                   │
│                                   │
│        DCB SKELETON SECTION       │
│                                   │
│                                   │
│                                   │
└───────────────────────────────────┘
```

Unit Block Sections

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

HOUSEKEEPING SECTION: This section is maintained by IHCFIOSH. The information contained in it is used to make various types of checks, 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 described below.

Data Set Type (ABYTE) Field: This field, containing the data set type passed to IHCFIOSH 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

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

0 -- exit to (IHCFCOMH) on input error
1 -- error occurred
2 -- current buffer indicator
3 -- not used
4 -- end of current buffer indicator
5 -- blocked data set indicator
6 -- variable RECFM switch
7 -- not used

CBYTE Indicator Field: This field also contains bits that are set and examined by IHCFIOSH. The bits and their usages are outlined below.

BIT ON

0 -- data set open
1 -- data set not TCLOSEd
2 -- data set previously opened
3 -- buffer pool attached
4 -- data set not previously rewound
5 -- data set not previously backspaced
6 -- concatenation occurring - reread
7 -- not used

LIVECNT Field: 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.

Buffer 1 and Buffer 2 Fields: These fields are filled with pointers to the I/O buffers obtained during the opening of the data set (performed by the OPEN or GETPOOL macro-instruction).

Block Pointer Field: This field contains a pointer to the I/O buffer currently in use by IHCFCOMH.

Record Offset Field: This field contains a pointer (within the current buffer) to the next logical record.

DECB SKELETON: The DECB (Data Event Control Block) skeleton is a block of storage within the unit block. It is of the same form as the DECB constructed by the control programs for an L form of an S-type READ or WRITE macro-instruction (refer to IBM System/360 Operating System: Introduction to Control Program Logic, Program Logic Manual). The various fields of the skeleton are filled in by IHCFIOSH; the completed skeleton is referred to by IHCFIOSH when it issues a read/write request to BSAM. For each I/O operation, IHCFIOSH supplies an indication of the type of operation (read or write), and the length of and a pointer to the I/O buffer to be used. IHCFIOSH also inserts the addresses of the associated DCB skeleton into the DECB.

DCB SKELETON: The DCB (Data Control Block) skeleton is a block of storage within the unit block. It is of the same form as the DCB constructed by the control programs for a DCB macro-instruction under BSAM (refer to IBM System/360 Operating System: Introduction to Control Program Logic, Program Logic Manual). Its various fields are filled in by the control programs when the data set is opened or by IHCFIOSH (see "Default Values") at DCB exit time.

UNIT ASSIGNMENT TABLE

The unit assignment table, consisting of two entries for each unit number, resides within IHCFIOSH. The first entry for a particular unit number contains either (1) a pointer to the unit block generated for that unit number, or (2) a value of 0 (indicating that a unit block has not been generated for the associated unit number). The second entry contains the default values for the unit (see "Default Values").

The unit assignment table is used as an index to the unit blocks. The first reference to a particular unit number causes IHCFIOSH to generate a unit block for that number. The address of the unit block is placed into the first entry in the unit assignment table for the unit number. All subsequent references to the unit number are then made through the unit assignment table.

Default Values

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

- Causes the object module to be executed via a cataloged procedure.

- In stating his own procedure for execution, fails to include in the DCB parameters of his DD statements, those subparameters (e.g., LRECL) he is permitted to include (see IBM Operating System/360: FORTRAN IV Programmer's Guide).

Note: Control is returned to IHCFIOSH during data set opening so that it can determine if the user has included these subparameters in the DCB parameter. (If the user has included these subparameters, the control program performing data set opening inserts the subparameter values, before giving control to IHCFIOSH, into the DCB skeleton fields reserved for those values.) IHCFIOSH examines the DCB skeleton fields corresponding to the used-permitted subparameters, and inserts the standard values (i.e., default values) for the non-specified subparameters into the DCB skeleton.

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 set opening, two buffers are 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 (see IBM Operating System/360: Control Program Services).

OPERATION

The processing of IHCFIOSH is divided into five sections: initialization, read, write, device manipulation, and closing. When called upon by IHCFCOMH, a section performs its function and then returns control to IHCFCOMH. The overall logic of IHCFIOSH is illustrated in Chart 27.

INITIALIZATION

The initialization action taken by IHCFIOSH depends upon the nature of the previous I/O operation. The operation possibilities are:

- No previous operation.
- Previous operation read/write.
- Previous operation backspace.
- Previous operation write end-of-data set or read taking "END=" exit.
- 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 that of 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 that of 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.

Previous Operation Read/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 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 or Read Taking "END=" Exit

If the previous operation performed on the unit specified in the present I/O request was either that of the write end-of-data set or that of read taking the "END=" exit, 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 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:

- Reads physical records into the buffers obtained during data set opening.
- Makes the contents of these buffers available to IHCFCOMH for processing.

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

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

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). 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 buffers contents is initiated. In addition, a read into the second buffer is also initiated.

WRITE

The write section of IHCFIOSH performs two functions:

- Provides IHCFCOMH with buffer space.
- Writes physical records.

Each time this section is given control, it provides IHCFCOMH with buffer space in which to place the record to be written. It places a pointer to the location within the current buffer at which IHCFCOMH is to place the record and the block size or logical record length into registers, and returns control to IHCFCOMH.

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

The writing of records by this section is also overlapped. While IHCFCOMH is filling one buffer, the contents of the other buffer is being written.

DEVICE MANIPULATION

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

Backspace

IHCFIOSH processes the backspace request by issuing a BPS macro-instruction (physical BACKSPACE). 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. Under certain conditions, IHCFIOSH 'forces' the data set type.)

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, Type=T, macro-instruction. It then frees the I/O buffers by issuing a FREEPOOL macro-instruction, and returns control to IHCFCOMH.

CLOSING

The function of the closing section of IHCFIOSH is to terminate I/O operations. It accomplishes this by examining the entries in the unit assignment table to determine which data sets are open. The closing section then checks (via the CHECK macro-instruction) all pending I/O operations on the open data sets, and returns control to IHCFCOMH.

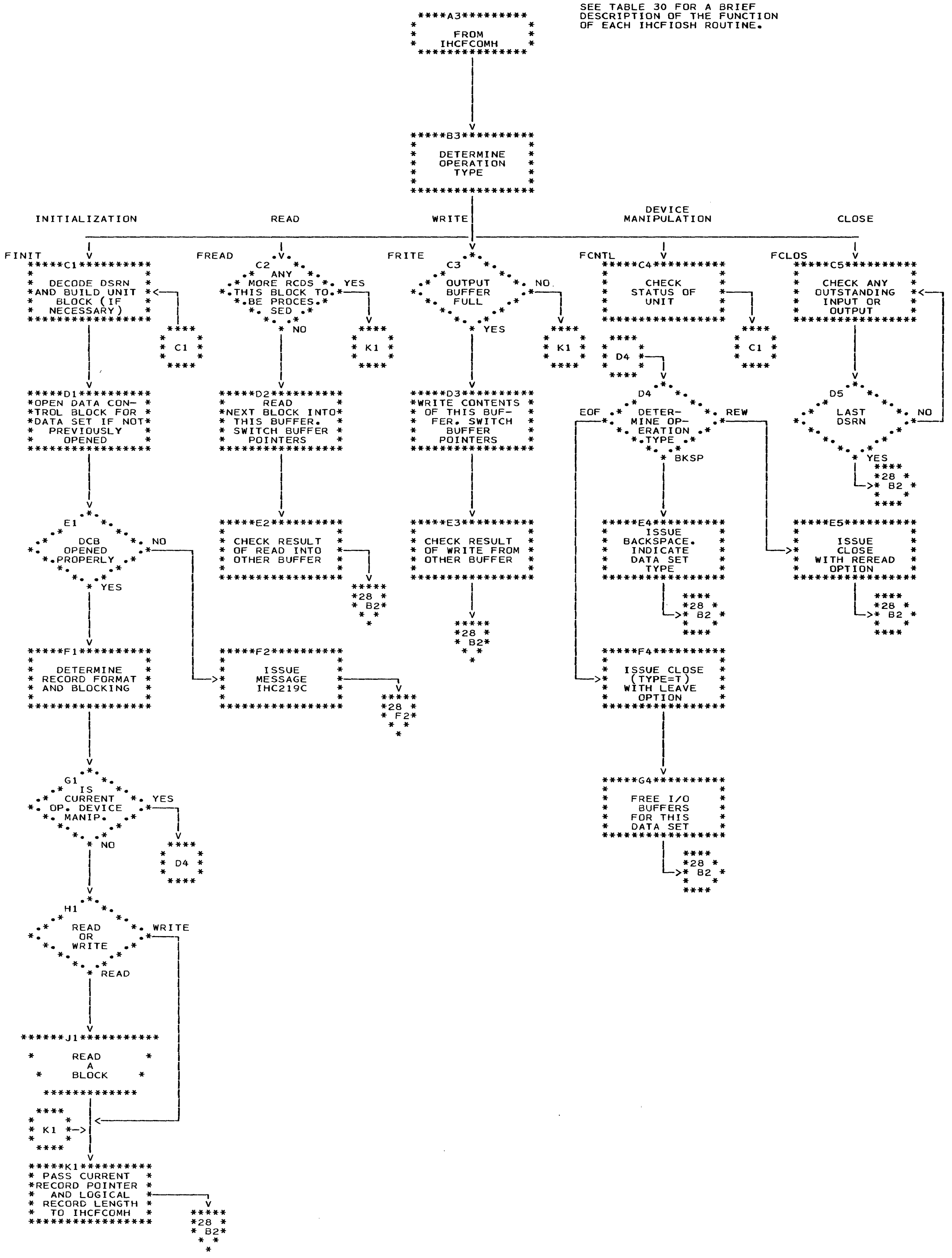
ERROR PROCESSING

If an end-of-data set or an I/O error is encountered during reading or writing, the control program returns control to IHCFIOSH. 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 this switch, finds it to be set, and passes control to the I/O error subroutine of IHCFCOMH.

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

Chart 27. IHCFIOSH Overall Logic

SEE TABLE 30 FOR A BRIEF DESCRIPTION OF THE FUNCTION OF EACH IHCFIOSH ROUTINE.



A3 FROM IHCFCOMH
B3 DETERMINE OPERATION TYPE
INITIALIZATION
READ
WRITE
DEVICE MANIPULATION
CLOSE
FINIT
C1 DECODE DSRN AND BUILD UNIT BLOCK (IF NECESSARY)
C1
D1 OPEN DATA CONTROL BLOCK FOR DATA SET IF NOT PREVIOUSLY OPENED
E1 DCB OPENED PROPERLY
NO
YES
F1 DETERMINE RECORD FORMAT AND BLOCKING
G1 IS CURRENT OP. DEVICE MANIP.
YES
D4
NO
H1 READ OR WRITE
WRITE
READ
J1 READ A BLOCK
K1
K1 PASS CURRENT RECORD POINTER AND LOGICAL RECORD LENGTH TO IHCFCOMH
28 B2
FREAD
C2 ANY MORE RCDS THIS BLOCK TO BE PROCESSED
YES
K1
NO
D2 READ NEXT BLOCK INTO THIS BUFFER. SWITCH BUFFER POINTERS
E2 CHECK RESULT OF READ INTO OTHER BUFFER
28 B2
F2 ISSUE MESSAGE IHC219C
28 F2
FRITE
C3 OUTPUT BUFFER FULL
NO
K1
YES
D3 WRITE CONTENTS OF THIS BUFFER. SWITCH BUFFER POINTERS
E3 CHECK RESULT OF WRITE FROM OTHER BUFFER
28 B2
FCNTL
C4 CHECK STATUS OF UNIT
C1
D4
D4 DETERMINE OPERATION TYPE
EOF
REW
BKSP
E4 ISSUE BACKSPACE. INDICATE DATA SET TYPE
28 B2
F4 ISSUE CLOSE (TYPE=T) WITH LEAVE OPTION
G4 FREE I/O BUFFERS FOR THIS DATA SET
28 B2
FCLOS
C5 CHECK ANY OUTSTANDING INPUT OR OUTPUT
D5 LAST DSRN
NO
YES
28 B2
E5 ISSUE CLOSE WITH REREAD OPTION
28 B2

Chart 28. Execution-Time I/O Recovery Prog

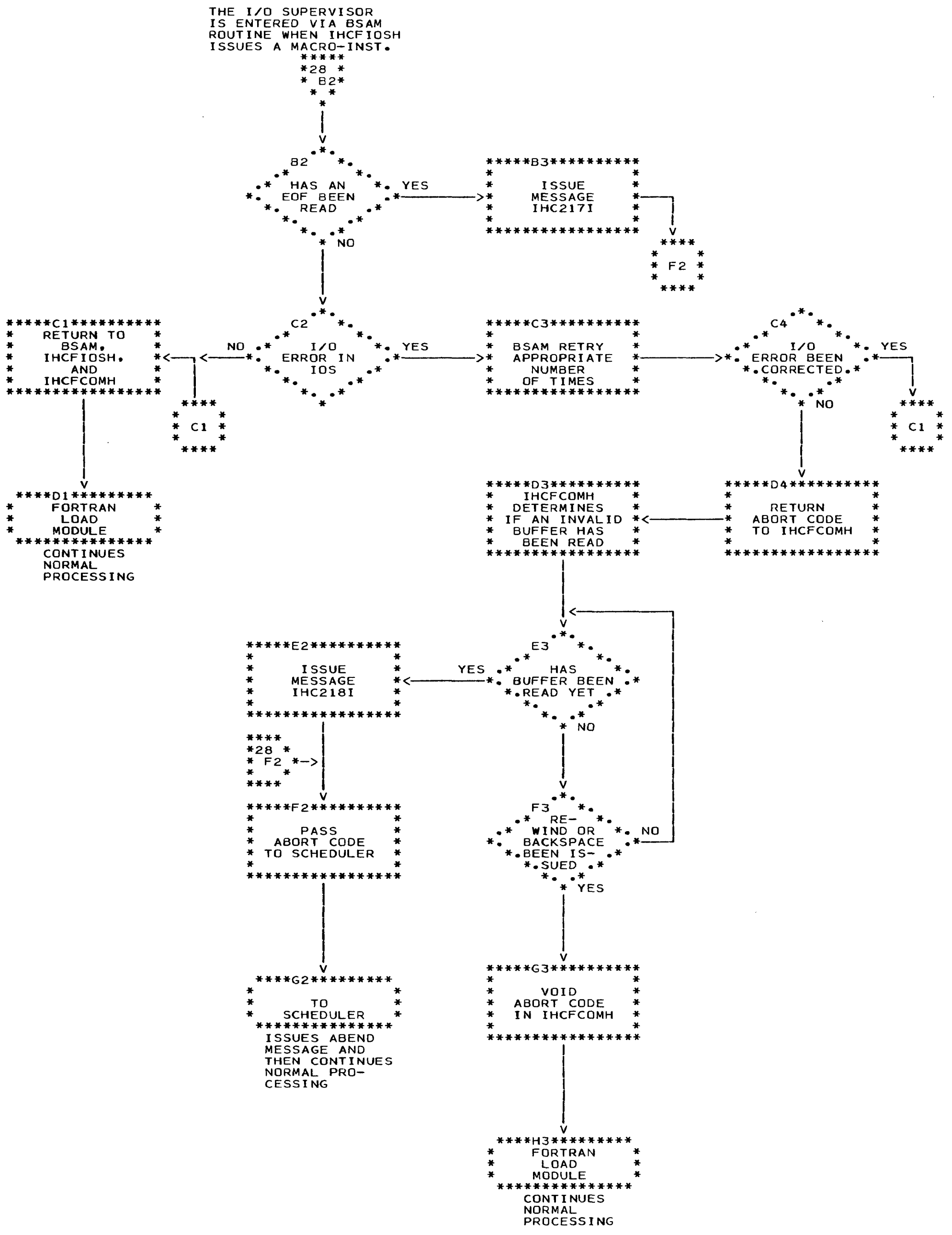

THE I/O SUPERVISOR IS ENTERED VIA BSAM ROUTINE WHEN IHCFIOSH ISSUES A MACRO-INST.
28 B2
B2 HAS AN EOF BEEN READ
YES
NO
B3 ISSUE MESSAGE IHC217I
F2
C1 RETURN TO BSAM, IHCFIOSH, AND IHCFCOMH
C2 I/O ERROR IN IOS
NO
YES
C1
C3 BSAM RETRY APPROPRIATE NUMBER OF TIMES
C4 I/O ERROR BEEN CORRECTED
YES
NO
C1
D1 FORTRAN LOAD MODULE
CONTINUES NORMAL PROCESSING
D3 IHCFCOMH DETERMINES IF AN INVALID BUFFER HAS BEEN READ
D4 RETURN ABORT CODE TO IHCFCOMH
E2 ISSUE MESSAGE IHC218I
E3 HAS BUFFER BEEN READ YET
YES
NO
28 F2
F2 PASS ABORT CODE TO SCHEDULER
F3 RE-WIND OR BACKSPACE BEEN IS-SUED
NO
YES
G2 TO SCHEDULER
ISSUES ABEND MESSAGE AND THEN CONTINUES NORMAL PRO-CESSING
G3 VOID ABORT CODE IN IHCFCOMH
H3 FORTRAN LOAD MODULE
CONTINUES NORMAL PROCESSING

Table 30. IHCFIOSH Subroutine Directory

Subroutine	Function
FCLOS	Terminates I/O operations.
FCTRL	Services device manipulation requests.
FINIT	Initializes unit and data set.
FREAD	Services read requests.
FRITE	Services write requests.

Data references in the form of subscripted variables 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 - (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

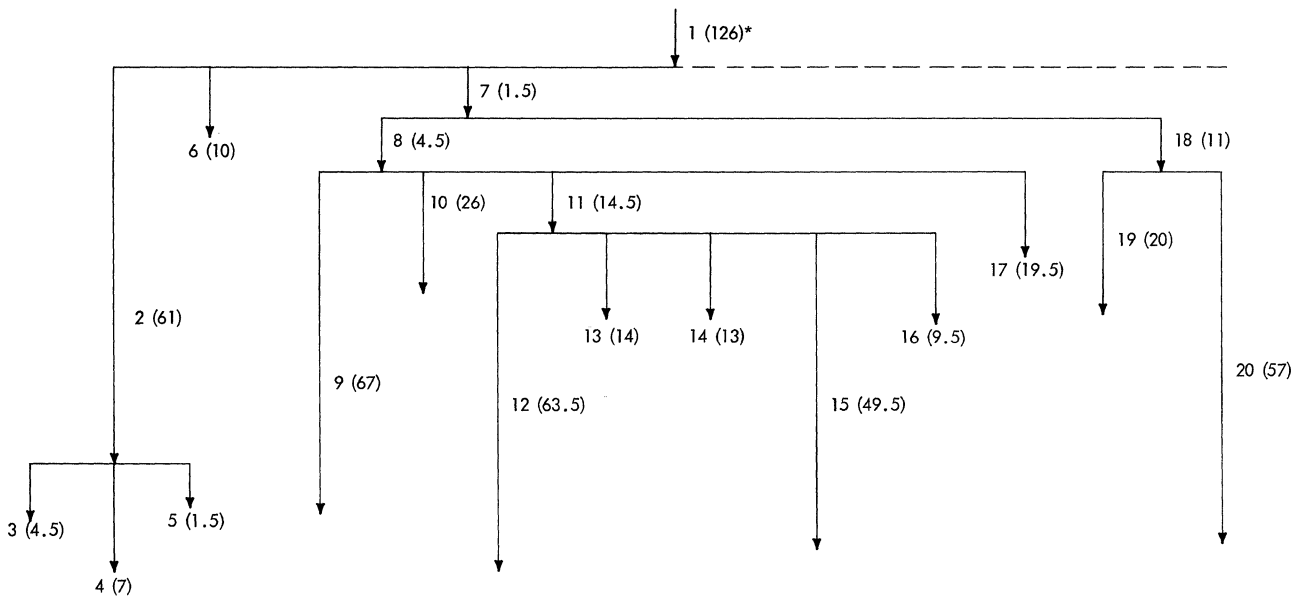
When an array is used as an argument, the location of its first element, P11, is passed in the parameter list. The prologue of the called subroutine contains machine code to compute the corresponding P00 location. When an array has variable dimensions, no constant absorption takes place and the dimension factors are computed for each reference to the array.

The FORTRAN IV (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 a planned overlay structure, and instruction 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 the 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 20 segments, one of which is the root. The root segment contains those processing units (e.g., the FSD) and data areas (e.g., communication region) that are used by two or more compiler phases. The root segment remains in main storage throughout execution of the compiler.

Each of the remaining 19 segments constitutes a phase, or a logical portion of a phase. Phase segments are overlayed as compiler processing requires the services of another segment.

Figure 61 illustrates the compiler's planned overlay structure. In the figure, each segment is identified by number. Segments associated with vertical line originating from the same horizontal line overlay each other as needed. The figure also indicates the approximate size (in bytes) of each segment.



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

Figure 61. Compiler Overlay Structure

The longest path[1] of this structure is formed by segments 1, 7, 8, 11, and 12, because, when they are in main storage, the compiler requires approximately 210,000 bytes. Thus, the minimum main storage requirement for the compiler is approximately 210,000 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 0 plus the length of all segments in the path. For example, the origin of segments 3, 4, and 5 is equal to 0 plus the length of segment 1 plus the length of segment 2.

The segments that constitute each of the compiler phases are outlined in Table 31. The remainder of this appendix is devoted to a discussion of the segments of the compiler's planned overlay structure.

Table 31. Phases and Their Segments

Phase	Segment(s) Constituting Phase
Phase 10	Segments 2, 3, 4, and 5
Phase 15	Segments 6, 7, 8, 9, and 10
Phase 20	Segments 8, 11, 12, 13, 14, 15, and 16
Phase 25	Segments 18, 19, and 20
Phase 30	Segment 17
Note: Segment 8 is considered a portion of both Phases 15 and 20. It contains data areas used by both phases.	

Segment 1: This segment is the root of the compiler's planned overlay structure. Segment 1 is the FSD. It has a relocatable origin at zero. Segment 1 is not overlayed. The composition of segment 1 is illustrated in Table 32.

---

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

Table 32. Segment-1 Composition

Control Section Name	Entry Point(s)
$SEGTAB	
BLANK	
ADCON	
ERCOM	
IEKFCOMH	IBCOM
	IBCOM#
IEKFIOCS	FIOCS
	FIOCS#
IEKAA01	
$BLANKCOM	
AFRXPI	FRXPI#
	FRXPI
SYSTAB	SYSTAB
IEKAA00	GETCOR
	ENDFILE
	SYSDIR
	PAGE
IEKUATPT	
SYSTRC	SYSTRC
IHCFMAXI	MAX0
	MIN0
	AMAX0
	AMIN0
IHCFMAXR	MAX1
	MIN1
	AMAX1
	AMIN1
$ENTAB	

Segment 2: This segment is a portion of phase 10. It contains the preparatory, dispatch, and utility subroutines of phase 10. It also contains a portion of the key word and arithmetic subroutines. (The arithmetic subroutines that perform the initial and final processing of statement functions are not in this segment.) Segment 2 is common to segments 3, 4, and 5, and its origin is immediately after segment 1. (A segment common to two or more segments is part of the path of each segment.) Segment 2 is overlayed by segment 6. The composition of segment 2 is illustrated in Table 33.

Table 33. Segment-2 Composition

Control Section	Entry Point(s)
PH10	
XARITH	XARITH
XCLASS	XCLASS
P10A	
GETWD	
GENDO	For GENDO and the remaining control sections of this segment (except for $ENTAB), the control section names and the entry point names are the same.
XCONT	
ERROR	
XSTOP	
RTPRQT	
XPUSE	
LITCON	
GETCD	
XGO	
XEQUI	
DSPTCH	
XNMLST	
CSORN	
GRPKEQ	
PERLOG	
XDO	
CDOPAR	
XDATA	
XBCKRW	
XIMPD	
SYMTLU	
XEXT	
XFMT	
LABTLU	
MINSLS	
XEND	
XIF	
CLOSE	
COMAST	
COMPAT	
INTCON	
PUTX	
TXTBLD	
XIOOP	
XRETN	
$ENTAB	

Segment 3: This segment is a portion of phase 10. It contains several key word subroutines. Segment 3 depends upon segment 2 in that it requires the use of certain utility subroutines of that segment. The origin of segment 3 is immediately after segment 2. Segment 3 overlays, and can be overlayed by, either segment 4 or segment 5. The composition of segment 3 is illustrated in Table 34.

Table 34. Segment-3 Composition

Control Section	Entry Point(s)
XSUBPG	For this segment, control section names and entry point names are the same.
XBLOK	
XIMPC	

Segment 4: This segment is a portion of phase 10. It contains several key word subroutines and those arithmetic subroutines that perform the initial and final processing of statement functions. Segment 4 depends upon segment 2 for utility services, and its origin is immediately after segment 2. Segment 4 overlays, and can be overlayed by, either segment 3 or segment 5. The composition of segment 4 is illustrated in Table 35.

Table 35. Segment-4 Composition

Control Section	Entry Point(s)
XTYPE	For this segment, control section names and entry point names are the same.
XDIM	
XCOMON	
XASF	
XASF2	

Segment 5: This segment is a portion of phase 10. It consists of several key word subroutines and is dependent upon segment 2 for utility services. The origin of segment 5 is immediately after segment 2. Segment 5 overlays, and can be overlayed by, either segment 3 or segment 4. The composition of segment 5 is illustrated in Table 36.

Table 36. Segment-5 Composition

Control Section	Entry Point(s)
XASGN	XASGN
XSTRUC	XSTRUC

Segment 6: This segment is a portion of phase 15. It contains the subroutines that sort the dictionary, and process COMMON and EQUIVALENCE declarations. The origin of segment 6 is immediately after segment 1 (the root segment). Segment 6 overlays segment 2, and is overlayed by segment 7. The composition of segment 6 is illustrated in Table 37.

Table 37. Segment-6 Composition

Control Section	Entry Point(s)
LABSCN	For this segment, control section names and entry point names are the same.
DCTSRT	
COMN	
EQU	
SBEROR	
STALL	
BSIZE	
TESTBN	

Segment 7: This segment is a portion of phase 15. It contains the subprogram table (IFUNTB), which is used by both the PHAZ15 and CORAL segments of phase 15. The origin of segment 7 is immediately after segment 1. Segment 7 overlays segment 6. The composition of segment 7 is illustrated in Table 38.

Table 38. Segment-7 Composition

Control Section	Entry Point(s)
FUNTB	

Segment 8: This segment is considered a portion of both phases 15 and 20. It contains data areas that are used by both these phases. Included in this segment are RMAJOR, CMAJOR, the full register assignment tables, and phase 15/20 work areas. The origin of segment 8 is immediately after segment 7. Segment 8 is overlaid by segment 18, if abortive errors are not encountered during the processing of phases 10 and 15. The composition of segment 8 is illustrated in Table 39.

Table 39. Segment-8 Composition

Control Section	Entry Point(s)
C1520	

Segment 9: This segment is a portion of phase 15, It contains the subroutines that implement the PHAZ15 functions of that phase, which are arithmetic translation, text blocking, and information gathering. The origin of segment 9 is immediately after segment 8. Segment 9 is overlayed by segment 10. The composition of segment 9 is illustrated in Table 40.

Table 40. Segment-9 Composition

Control Section	Entry Point(s)
SUBSCR	SUBSCR
PH15	
MATE	MATE
STTEST	STTEST
BLTNFN	BLTNFN
DUMP15	DUMP15
EXPON	EXPON
ANDOR	ANDOR
CPLTST	CPLTST
PHAZ15	PHAZ15
SUBMLT	SUBMLT
GENRTN	GENRTN
LOOKER	
ALTRAN	ALTRAN
MODTST	MODTST
XPARAM	XPARAM
DFUNCT	DFUNCT
RELOPS	RELOPS
FINISH	FINISH
PAREN	PAREN
LIBRTN	LIBRTN
TXTREG	TXTREG
GENER	GENER
RDTST	RDTST
GETEXT	GETEXT
ARIF	ARIF
NEGCHK	NEGCHK
UNARY	UNARY
GMAT	GMAT
TXTLAB	TXTLAB
VSETUP	VSETUP
WRIT15	WRIT15
MNE	
SBGLUT	SBGLUT
FUNDRY	FUNDRY
SUBADD	SUBADD
MODIFY	MODIFY
NOT	NOT
OP1CHK	OP1CHK
POWER2	POWER2
COMMD	COMMD
NSTRNG	NSTRNG
SWITCH	SWITCH

Segment 10: This segment is a portion of phase 15. It contains the subroutines that implement the CORAL functions of the phase. The origin of segment 10 is immediately after segment 8. Segment 10 overlays segment 9. Segment 10 is overlaid by segment 11, if syntactical errors are not encountered by phase 10 and 15. If errors are present, segment 10 is overlaid by segment 17. The composition of segment 10 is illustrated in Table 41.

Table 41. Segment-10 Composition

Control Section	Entry Point(s)
EXTRNL	EXTRNL
STMAP2	
NDATA	For NDATA and the remaining control sections of this segment, the control section names and entry point names are the same.
VARA	
CORAL	
TESTWD	
EQVAR	
CONST	
CMSIZE	
COMVAR	
ADSCAN	
DATACH	
ERDATA	
SIZE	
PRTEXT	
SPAN	
CORLDT	

Segment 11: This segment is a portion of phase 20. It contains the controlling subroutine of that phase, the loop selection routines, and a number of frequently used utility subroutines. The origin of segment 11 is immediately after segment 8. Segment 11 overlays segment 10, if source module errors are not encountered by phases 10 and 15. If errors are encountered, segment 11 overlays segment 17 after its processing is completed, only if the errors encountered are not serious enough to cause the deletion of the compilation. The composition of segment 11 is illustrated in Table 42.

Table 42. Segment-11 Composition

Control Section	Entry Point(s)
CNT	
OPT	
GETDIK	GETDIK
GETDIC	GETDIC
LPSEL	LPSEL
NPRFUN	NPRFUN
INVERT	INVER
GETSPC	GETSPC
FILTEX	FILTEX
TARGET	TARGET
BASVAR	BASVAR
BSYONX	BSYONX

Segment 12: This segment is a portion of phase 20. It contains the text optimization subroutines and the utility subroutines used by them. Segment 12 is executed only if the complete-optimized path through phase 20 is specified. The origin of segment 12 is immediately after segment 11. During the course of complete optimization, segment 12 overlays segment 14. Segment 12 is overlayed by segment 15 after all module loops have been text-optimized. The composition of segment 12 is illustrated in Table 43.

Table 43. Segment-12 Composition

Control Section	Entry Point(s)
NORMIZ	NORMIZ
MOV	
REDUCE	REDUCE
MOZ	
PARFIX	For PARFIX and the remaining control sections, the control section names and entry point names are the same.
SUBACT	
YCHANG	
CLASIF	
BACMOV	
SUBTRY	
GROUPA	
GROUPC	
GROUPB	
SUBSUM	
ZCHANG	
PERTRY	
MODFIX	
LORAN	
PERFOR	
MOVTEX	
OBTAIN	
XSCAN	
XPLACE	
YPLACE	
YSCAN	
ZPLACE	
ZSCAN	
TAGLOC	
MBRAN	
AGGLUT	
CIRCLE	
DELTEX	
XCHANG	
XPELIM	
KORAN	
FOLLOW	
XPELOC	
TYPLOC	
WRITEX	
FORMOV	
INDTRY	
INERT	

Segment 13: This segment is a portion of phase 20. It consists of the subroutines that perform basic register assignment. Segment 13 is only executed in the non-optimized path through phase 20. The origin of segment 13 is immediately after segment 11. Segment 13 does not overlay any other segment in phase 20, nor is it overlaid by another segment in phase 20. The composition of segment 13 is illustrated in Table 44.

Table 44. Segment-13 Composition

Control Section	Entry Point(s)
TALL	TALL
SPLRA	SPLRA
SSTAT	SSTAT
STDMP	STDMP

Segment 14: 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 14 is only executed if the complete-optimized path through phase 20 is followed. This segment is only executed once and is overlaid by segment 12. The origin of segment 14 is immediately after segment 11. The composition of segment 14 is illustrated in Table 45.

Table 45. Segment-14 Composition

Control Section	Entry Point(s)
BAKT	BAKT
BLK	
SRPRIZ	SRPRIZ
TOPO	TOPO
BIZX	BIZX

Segment 15: This segment is a portion of phase 20. It contains full register assignment subroutines and the utility subroutines used by them. Segment 15 is executed in both the intermediate-optimized and complete-optimized paths through phase 20. In the intermediate-optimized path, segment 15 is overlayed by segment 16. During complete-optimization, segment 15 overlays segment 12 after all loops have been text-optimized and is overlayed by segment 16 after all loops have undergone full register assignment. The origin of segment 15 is immediately after segment 11. The composition of segment 15 is illustrated in Table 46.

Table 46. Segment-15 Composition

Control Section	Entry Point(s)
REGAS	REGAS
REG	
PROP1	PROP1
BKP	
LOC	
FWDPAS	FWDPAS
FWP	
BKPAS	BKPAS
GTBASE	GTBASE
ALLCOR	ALLCOR
STX	
GLOBAS	GLOBAS
FCLT50	FCLT50
STXTR	STXTR
GLS	
MRCLEN	For MRCLEN and the remaining control sections, the control section names and entry point names are the same.
CXIMAG	
FWDPS1	
HILOWS	
SETUP	
GLOBS1	
ACCEPT	
DISCHK	
SEARCH	
FREE	
SHARE	
TRNSFM	
SETREG	
RELCOR	
PRELUD	
BKDMP	

Segment 16: This segment is a portion of phase 20. It consists of the subroutines that 1) calculate the size of each text block and 2) determine which text blocks can be branched to via RX-format branch instructions. Segment 17 is executed in both the intermediate-optimized and complete-optimized paths. Segment 16 overlays segment 15 after full register assignment is completed. Segment 16 is not overlayed within phase 20. The origin of segment 16 is immediately after segment 11. The composition of segment 16 is illustrated in Table 47.

Table 47. Segment-16 Composition

Control Section	Entry Point(s)
SEG4	SEG4
BLS	BLS
LYT	LYT
BLSDTA	
BSTRIP	

Segment 17: This segment is phase 30. The origin of segment 17 is immediately after

segment 8. Segment 17 overlays segment 10, 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 17, after its processing is completed, is overlaid by segment 11. The composition of segment 17 is illustrated in Table 48.

Table 48. Segment-17 Composition

Control Section	Entry Points(s)
IEKP30	IEKP30
MSGWRT	MSGWRT

<u>Segment 18:</u> This segment is a portion of phase 25. It contains a number of subroutines that are employed by both the initial text information construction and the text conversion portions of phase 25 (see Charts 21 and 22). The origin of segment 18 is immediately after segment 7. Segment 18 overlays segment 8. The composition of segment 18 is illustrated in Table 49.

Table 49. Segment-18 Composition

Control Section	Entry Points(s)
FAZ25	
BXHCOM	
PROLOG	PROLOG
DCLIST	DCLIST
LISTER	LISTER
END	END
LABEL	LABEL
IEKTLOAD	ESD
	TXT
	RLD
	IEND
INITIA	INITIA
PACKER	PACKER
EPILOG	EPILOG
$ENTAB	

<u>Segment 19:</u> This segment is a portion of phase 25. It contains most of the subroutines that perform initial text information construction (see Chart 21.) The origin of segment 19 is immediately after segment 18. Segment 19 is overlaid by segment 20. The composition of segment 19 is illustrated in Table 50.

Table 50. Segment-19 Composition

Control Section	Entry Point(s)
NADOUT	For this segment,
SUBR	the control section
ATTACH	names and entry
FORMAT	point names are the
INITIL	same.
LYT1	
DATOUT	
NLIST	

<u>Segment 20:</u> This segment is a portion of phase 25. It contains the subroutines that perform text conversion (see Chart 22). The origin of segment 20 is immediately after segment 18. Segment 20 overlays segment 19. The composition of segment 20 is illustrated in Table 51.

Table 51. Segment-20 Composition

Control Section	Entry Points(s)
MANGN2	MANGN2
DBLGEN	DBLGEN
IOSUB	IOSUB
LBITTF	LBITTF
BRCOMB	BRCOMB
FLTGEN	FLTGEN
DIMGEN	DIMGEN
TSTSET	TSTSET
NTFXGN	NTFXGN
RETURN	RETURN
DIVGEN	DIVGEN
MAINGN	MAINGN
CGEN	
STRGEN	For STRGEN and the remainder of this segment, the control section names and entry point names are the same.
SHFT2	
IOSUB2	
CALLER	
IEKWAG	
TENTXT	
LDADDR	
BRCOMP	
STOPPR	
BRLGL	
BRANCH	
BTBF	
LGLNOT	
LDBGEN	
ENTRY	
SIGNGN	
ABSGEN	
GOTOKK	
LSTGEN	
SUBGEN	
MXMNGN	
LOGCL	
FNCALL	
CMPLGN	
ADMDGN	
NDORGN	
MOD24	
BITNFP	
SHFTRL	
PLSGEN	
MINUS	
INTMPY	
UNRGEN	
MODGEN	

The messages produced by the compiler are explained in the publication IBM System/360 Operating System: FORTRAN IV 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.

Message number	Routine in which message number is generated	Phase in which message number is generated
2	XCLASS	PHASE 10
3	PERLOG	
4	PERLOG	
5	RTPRQT	
7	MINSLS	
8	LITCON	
9	LITCON	
10	LITCON	
12	CSORN	
13	PUTX	
14	INTCON	
15	CDOPAR	
16	XGO	
17	XGO	
18	XGO	
19	XGO	
20	XGO	
21	XGO	
22	XGO	
27	XASGN	
29	RTPRQT	
30	XDO	
31	CDOPAR	
32	XARITH	
33	XARITH	
36	DSPTCH	
37	XASF	
40	PERLOG	

Message number	Routine in which message number is generated	Phase in which message number is generated
43	COMAST	
44	COMAST	
45	COMAST	
46	XDIM	
47	COMAST	
48	XARITH	
49	LITCON	
50	RPTRQT	
51	RPTRQT	
52	GRPKEQ	
53	GRPKEQ	
54	GRPKEQ	
55	GRPKEQ	
57	XSUBPG	PHASE 10
58	XSUBPG	
59	XSUBPG	
60	XARITH	
64	XNMLST	
65	XNMLST	
66	XNMLST	
67	XNMLST	
68	XCOMON	
69	XCOMON	
70	XEQUI	
71	XEQUI	
72	XEQUI	
73	XEQUI	
74	XEQUI	
75	XCOMON	

Message number	Routine in which message number is generated	Phase in which message number is generated
76	XARITH	
77	XIMPC	
78	XIMPC	
79	XIMPC	
80	XIMPC	
81	XIMPC	
82	XIMPC	
83	XIMPC	
84	XIMPC	
85	INTCON	
89	XEXT	
91	XEXT	
92	XTYPE	
93	XTYPE	
94	XTYPE	PHASE 10
95	XTYPE	
96	XTYPE	
97	XSTRUC	
98	XSTRUC	
100	XSTRUC	
102	XBCKRW	
103	XBCKRW	
104	XBCKRW	
105	XCONT	
106	XCONT	
107	XSTOP	
109	XPUSE	
110	XPUSE	
111	XPUSE	

Message number	Routine in which message number is generated	Phase in which message number is generated
112	XDATA, SYMTLU, XPUSE, LABTLU	PHASE 10
113	XRETN	
115	XRETN	
117	XBLOK	
120	XBLOK	
121	XDATA	
122	XDATA	
123	XDATA	
124	XDATA	
125	XDATA	
127	XDATA	
128	CDOPAR	
129	XDATA	
130	XDATA	
132	XDATA	
133	XDO	
134	XDO	
135	CDOPAR	
139	PERLOG	
140	XFMT	
141	XFMT	
142	XASF	
143	XASF	
144	XASF	
145	XASF	
146	XASF	
149	XDIM	
150	XDIM	
151	XDIM	
152	XSUBPG	
156	XIOOP	

Message number	Routine in which message number is generated	Phase in which message number is generated
158	XIMPD	PHASE 10
159	XFMT	
160	XIOOP	
161	XIOOP	
162	XIOOP	
163	XIMPD	
164	XIOOP	
165	XIOOP	
176	XIMPD	
193	XCLASS	
194	XTYPE	
197	XSTOP	
199	XSUBPG	
200	XDIM	
201	RTPRQT	
222	LITCON	
224	XCLASS	
229	XASF2	
302	EQU	PHASE 15 (STALL and CORAL)
304	EQU	
306	EQU	
308	EQU	
310	EQU	
312	EQU	
314	TESTBN	

Message number	Routine in which message number is generated	Phase in which message number is generated
318	NDATA	PHASE 15 (STALL and CORAL)
322	TESTBN	
332	LABSCN	
334	COMN	
350	NDATA	
352	NDATA	
353	EXTRNL	
354	CMSIZE	
355	CMSIZE	
500	FORMAT	PHASE 15 (PHAZ15)
501	EXPON	
502	EXPON	
509	PHAZ15	
510	ANDOR	
511	NOT	
512	FINISH	
515	RELOPS	
520	ALTRAN	
521	ALTRAN	
522	ALTRAN	
524	ALTRAN	
523	ALTRAN	
525	ALTRAN	
526	RELOPS	
527	ANDOR	
528	BLTNFN	
529	XPARAM	
530	SUBADD	
531	ALTRAN	
541	DFUNCT	
542	ALTRAN	
550	ALTRAN	PHASE 15 (PHAZ15)
551	GENER,GMAT	
552	GETEXT	
580	ALTRAN	
581	SUBMLT	
583	TXTREG	
584	MATE	
585	FINISH	
600	TOPO	PHASE 20
610	TOPO	
620	GETDIC	
621	GETDIK	
630	GETDIC	
631	GETDIK	
640	GETSPC	
650	TOPO	
660	TOPO	
670	BAKT	
671	BIZX	
680	RELCOR	
700	NADOUT	PHASE 25
710	FORMAT	
720	FORMAT	
730	FORMAT	
740	FORMAT	
750	FORMAT	
1000	IEKP30	PHASE 30
1	IEKP30	

INDEX

Y20-0012-0

Printed in U.S.A. Y20-0012-0

IBM®

International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, New York 10601