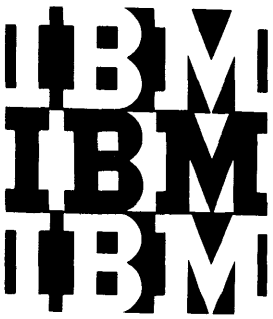


Y28-6601-1

Program Logic

IBM System/360 Operating System

FORTRAN IV (E)

Program Logic Manual

Program Number 360S-FO-092

This publication describes the internal design of the IBM System/360 Operating System FORTRAN IV (E) 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 manual is organized into three sections. Section 1 is an introduction and describes the overall structure of the compiler and its relationship to the operating system. Section 2 discusses the functions and logic of each phase of the compiler. Section 3 includes a series of flowcharts that show the relationship among the routines of each phase. Also provided in this section are phase routine directories.

Appendixes at the end of this publication provide information pertaining to: (1) source statement scan, (2) intermediate text formats, (3) table formats, (4) main storage allocation, etc.

Prerequisite to the use of this publication are:

IBM System/360 Operating System: Principles of Operation, Form A22-6821

IBM System/360 Operating System: FORTRAN IV (E) Language, Form C28-6513

IBM System/360 Operating System: Introduction to Control Program Logic, Program Logic Manual, Form Y28-6605

IBM System/360 Operating System: FORTRAN IV (E) Programmer's Guide, Form C28-6603 (sections "Job Processing" and "Cataloged Procedures")

Although not prerequisite, the following documents are related to this publication:

IBM System/360 Operating System: FORTRAN IV (E) Library Subprograms, Form C28-6596

IBM System/360 Operating System: Sequential Access Methods, Program Logic Manual, Form Y28-6604

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 Y28-6610

IBM System/360 Operating System: Data Management, Form C28-6537

IBM System/360 Operating System: System Generation, Form C28-6554

This compiler is similar in design to the IBM System/360 Basic Programming Support FORTRAN IV Compiler.

Second Edition
This is a major revision of, and obsoletes, Form Y28-6601-0 (formerly Z28-6601-0). Significant changes have been made throughout the text. This edition should be reviewed in its entirety.

Significant changes or additions to the specifications contained in this publication will be reported in subsequent revisions or Technical Newsletters.

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

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

A form is provided at the back of this publication for reader's comments. If the form has been removed, comments may be addressed to IBM Corporation, Programming Systems Publications, Department D58, PO Box 390, Poughkeepsie, N. Y. 12602

TABLES

CHARTS

The IBM System/360 Operating System FORTRAN IV (E) compiler analyzes source modules written in the FORTRAN IV (E) language and transforms them into object modules suitable for input to the linkage editor for subsequent execution on the IBM System/360. If the compiler detects errors in the source module, appropriate error messages are produced.

THE COMPILER AND OPERATING SYSTEM/360

The FORTRAN IV (E) compiler is a processing program of the IBM System/360 Operating System. As a processing program, the compiler communicates with the 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 batch of compilations, is introduced as a job step under the control of the operating system via the job statement (JOB), the execute statement (EXEC), and the data definition statements (DD) for the input/output data sets. To keep these statements at a minimum (in the input job stream), cataloged procedures are provided. A discussion of the introduction of a FORTRAN IV (E) compilation as a job step and of the available cataloged procedures is given in the publication IBM System/360 Operating System: FORTRAN IV (E) Programmer's Guide.

The compiler initially receives control from a calling program of the operating system (e.g., the initiator/terminator) by means of a supervisor-assisted linkage. Once the compiler receives control, it maintains communication with the operating system through:

- The interface module
- System macro-instructions

THE INTERFACE MODULE

The interface module, a component of the FORTRAN IV (E) compiler, resides on the operating system library (SYS1.LINKLIB). When the compiler receives control, it loads, via the LOAD macro-instruction, the interface module into main storage where it remains throughout the job step. The interface module processes all input/output requests of the compiler. The requests are initiated by a linkage to the interface module. The parameters necessary for I/O operations are passed to the interface module via this linkage. The interface module then links to the BSAM (basic sequential access method) read/write routine via the READ/WRITE macro-instruction. (A description of BSAM and the corresponding read/write routines is given in the publication IBM System/360 Operating System: Sequential Access Methods, Program Logic Manual.)

SYSTEM MACRO-INSTRUCTIONS

Whenever the XCTL, LOAD, DELETE, OPEN, CLOSE, READ, WRITE, CHECK, RDJFCB, GETMAIN, FREEMAIN, BLDL, SPIE, or TIME macro-instruction is issued, control is given directly to the operating system to execute the requested service.

When the job step (a single compilation or batch of compilations) is terminated, control is returned to the calling program via the RETURN macro-instruction.

COMPILER ORGANIZATION

The FORTRAN IV (E) compiler consists of several components, each of which exists as a separate load module on the operating system library (SYS1.LINKLIB). The components are:

- Phases (1, 7, 10D, 10E, 12, 14, 15, 20, 25, and 30).
- Interludes (10E, 14, and 15).
- Performance module.
- Interface module.
- Print buffer module.
- Source symbol module.
- Object listing module.

The compiler components and their major functions are shown in Table 1.

Table 1. Compiler Components and Their Major Functions

COMPONENT	MAIN FUNCTION(s)
Phase 1 (IEJFAAA0)	initializes compiler
Interface module (IEJFAGA0)	processes compiler I/O requests for all compilations, and end-of-phase/interlude requests for SPACE compilations
Print buffer module (IEJFAKA0)	contains two I/O buffers that are used for the SYSIN and SYSPRINT data sets
Performance module (IEJFAPA0)	reduces compilation time (loaded into main storage and executed only for PRFRM option); deblocks compiler input and blocks compiler output if blocking is specified; and processes end-of-phase requests for PRFRM compilations
Phase 7 (IEJFEAA0)	obtains and allocates main storage for resident tables and internal text buffers. (If the PRFRM option and blocking are specified, Phase 7 also obtains and allocates main storage for I/O buffers to be used by the block/deblock routine of the performance module.)
Phase 10D (IEJFGAA0)	transforms nonexecutable statements into intermediate text
Phase 10E (IEJFJAAO)	transforms executable statements into intermediate text
Interlude 10E (IEJFJGA0)	opens data control blocks required by Phases 12 and 14 (executed only for SPACE compilations)
Phase 12 (IEJFLAA0)	processes COMMON and EQUIVALENCE statements, and assigns relative addresses to variables and constants
Phase 14 (IEJFNAA0)	processes FORMAT and READ/WRITE statements
Interlude 14 (IEJFNGA0)	provides additional main storage for Phase 15 (executed only for SPACE compilations)
Phase 15 (IEJFPAA0)	processes arithmetic expressions
Interlude 15 (IEJFPGA0)	ensures that BSAM routines required in subsequent phases are present (executed only for SPACE compilations)
Phase 20 (IEJFRAA0)	optimizes subscript expressions
Phase 25 (IEJFVAA0)	generates object coding
Source symbol module (IEJFAXA0)	used by Phase 12 to contain the names of all variables and constants used in the source modules and their relative addresses (loaded into main storage only if the object listing option is specified and if the object listing facility is enabled)
Object listing module (IEJFVCA0)	used by Phase 25 to generate the object module listing (loaded into main storage only if the object listing option is specified and if the object listing facility is enabled)
Phase 30 (IEJFXAA0)	generates error/warning messages and processes the END statement
Phase 1 (IEJFAAB0)	terminates compilation (in the case of a batch compilation, Phase 1 performs transitional processing to initiate the next compilation)

COMMUNICATION AMONG COMPILER PHASES

When a compiler is divided into more than one phase, communication among the phases is required. Communication among the phases of the FORTRAN IV (E) compiler is implemented via:

- The communication area.
- Intermediate text.
- Resident tables.

THE COMMUNICATION AREA

The communication area (FCOMM) is a central gathering area (a portion of the interface module) for information common to the phases. It is used to communicate this information, when necessary, among the phases.

INTERMEDIATE TEXT

Source module statements (executable and nonexecutable) are converted into an internal text format (intermediate text). This intermediate text, once it is created, is used as input to the subsequent phases of the compiler. This text is eventually transformed into machine language instructions.

RESIDENT TABLES

The resident tables are the dictionary, the overflow table, the segment address list (SEGMAL), the patch table, the blocking table, and the BLDL table. The dictionary is a reference area containing information about variables, arrays, constants, and data set reference numbers used in the source module. The overflow table contains all dimension, subscript, and statement number information within the source module. SEGMAL is used for main storage allocation within the compiler. The patch table contains information to be used to modify compiler components. The blocking table contains information necessary for deblocking compiler input and blocking compiler output for PRFRM compilations. The BLDL table provides the information necessary for transferring control from one component to the next for PRFRM compilations. (The blocking table and the BLDL table reside in main storage only for PRFRM compilations.)

COMPILER CONTROL FLOW

If the SPACE option is specified by the user, control is passed among the components of the compiler via the interface module. After each component has been executed, that component branches to the interface module with the name of the component to be executed next. The interface module then issues an XCTL (transfer control) macro-instruction to the next component.

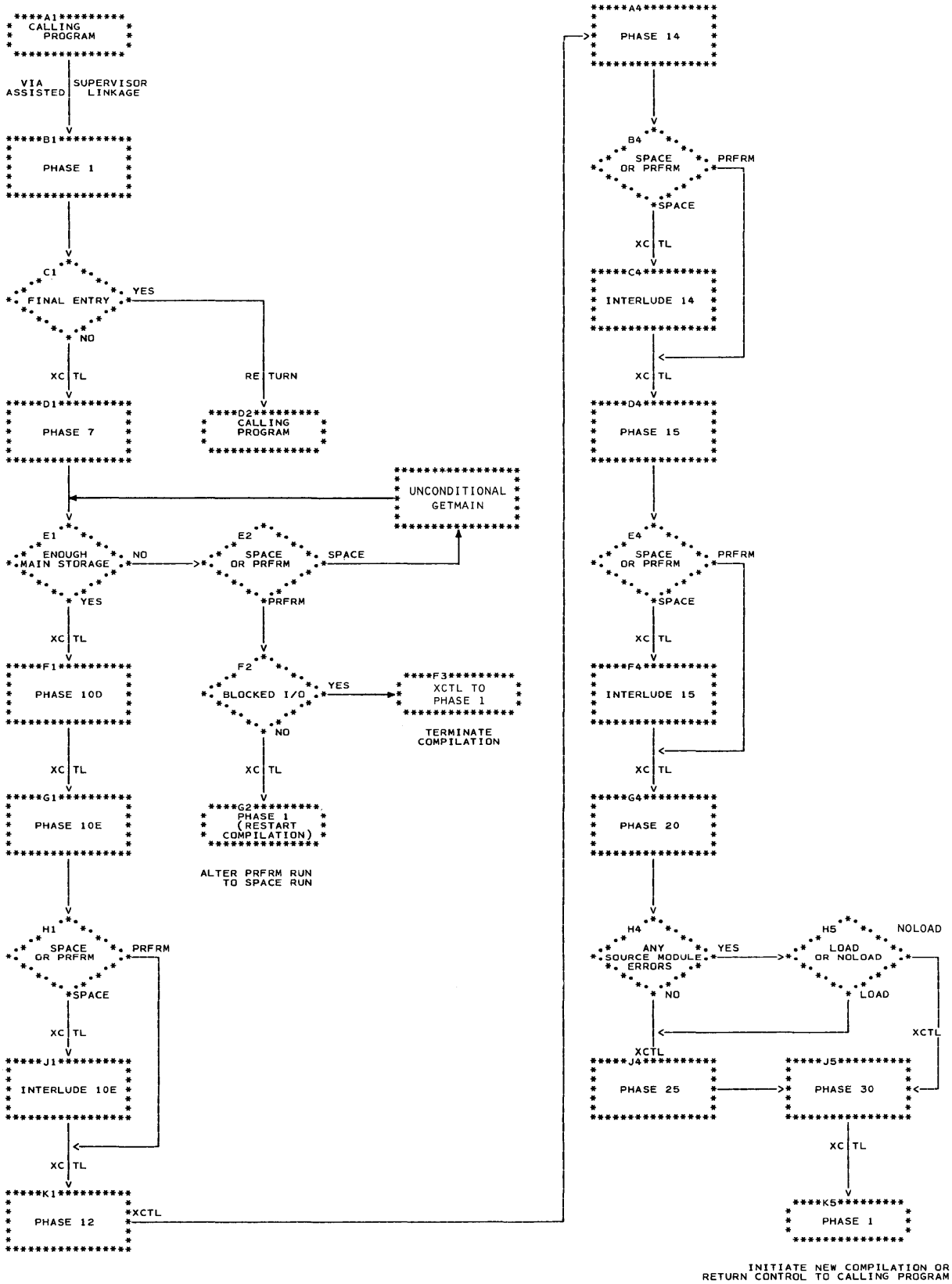
If the PRFRM option is specified by the user, control is passed among the components of the compiler via the performance module. After each component has been executed, that component branches to the performance module with the name of the component to be executed next. If the next component is an interlude, the performance module bypasses the execution of the interlude and transfers control, via the XCTL macro-instruction, to the next phase of the compiler. If the next component is a phase, the performance module immediately transfers control to that phase.

Note: The interludes are only executed if the SPACE option is specified by the user. (The SPACE option is chosen by the user if the amount of main storage that is available for compilation is limited.) Each interlude first closes the data control blocks for all the data sets that are open, and then opens only those for the data sets that are required by subsequent phases. This process decreases the size of the currently required BSAM routines and provides the additional main storage necessary to compile source modules in an environment in which the amount of available main storage is limited.

The performance module is loaded into main storage and executed only if the PRFRM option is specified by the user. (The PRFRM option is chosen by the user if he desires maximum compiler efficiency, and if the amount of available main storage is not a limitation.) A PRFRM compilation eliminates the execution of the interludes. The execution of the interludes can be bypassed because enough main storage is available to allow Phase 1 to initially open the data control blocks for all the data sets that are required for the entire compilation. The data control blocks are closed only at the end of the compilation. Bypassing the execution of the interludes decreases compilation time and therefore, increases overall compiler efficiency.

The overall compiler control flow is illustrated in Chart 00.

Chart 00. Overall Compiler Control Flow



COMPILER INPUT/OUTPUT FLOW

The source modules to be compiled are read into main storage by the compiler from the SYSIN data set. The compiler uses SYSUT1 and SYSUT2 as intermediate work data sets. (If the buffers used for reading and writing on these work data sets are large enough to contain the source module, then this data is retained in main storage.) The SYSLIN, SYSPRINT, and SYSPUNCH data sets are used for the output of the compilation. (SYSLIN is used only if the LOAD option is specified; SYSPUNCH is used only if the DECK option is specified.)

Figure 1 shows the compiler input/output structure.

Figure 2 shows the compiler input/output flow and includes intermediate input to and intermediate output from the various phases of the compiler.

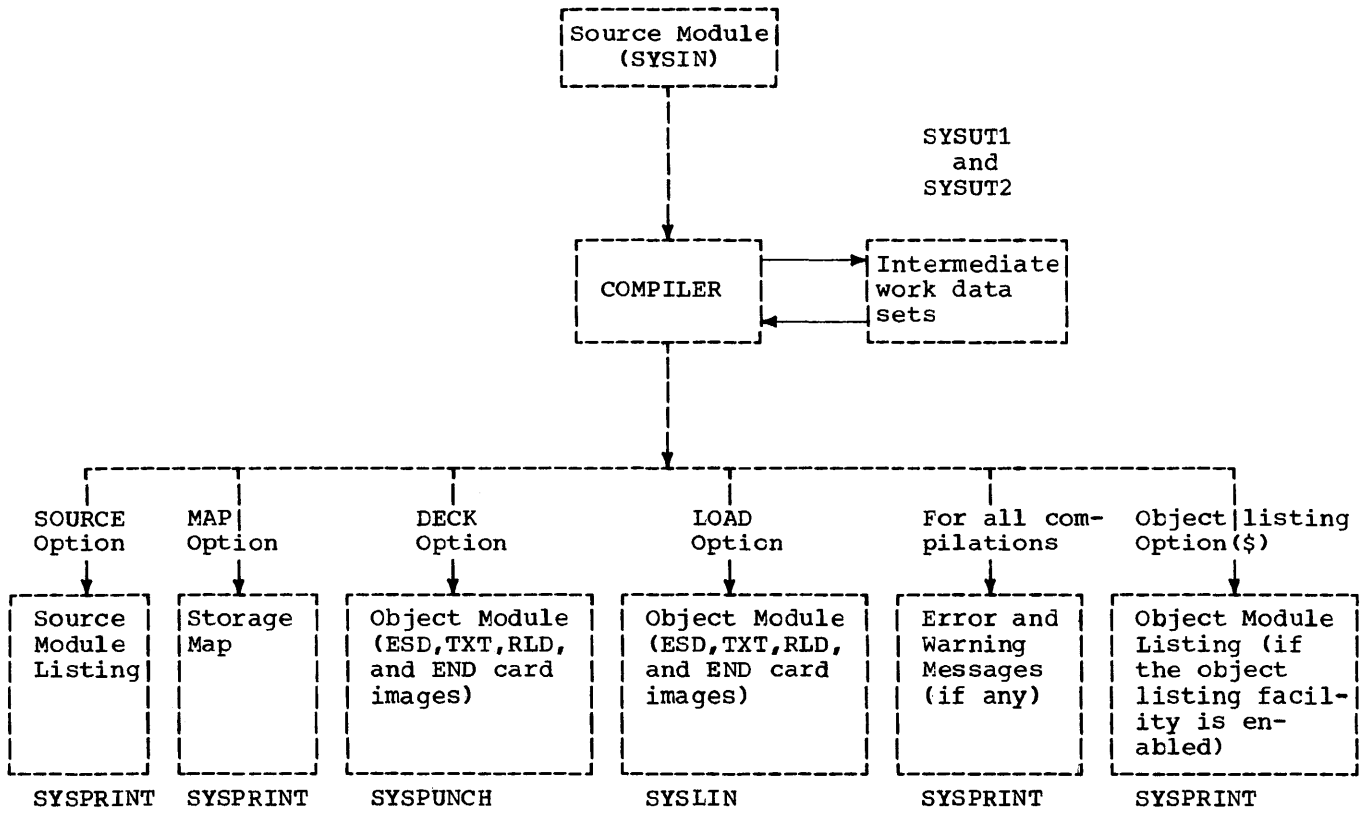


Figure 1. Compiler Input/Output Structure

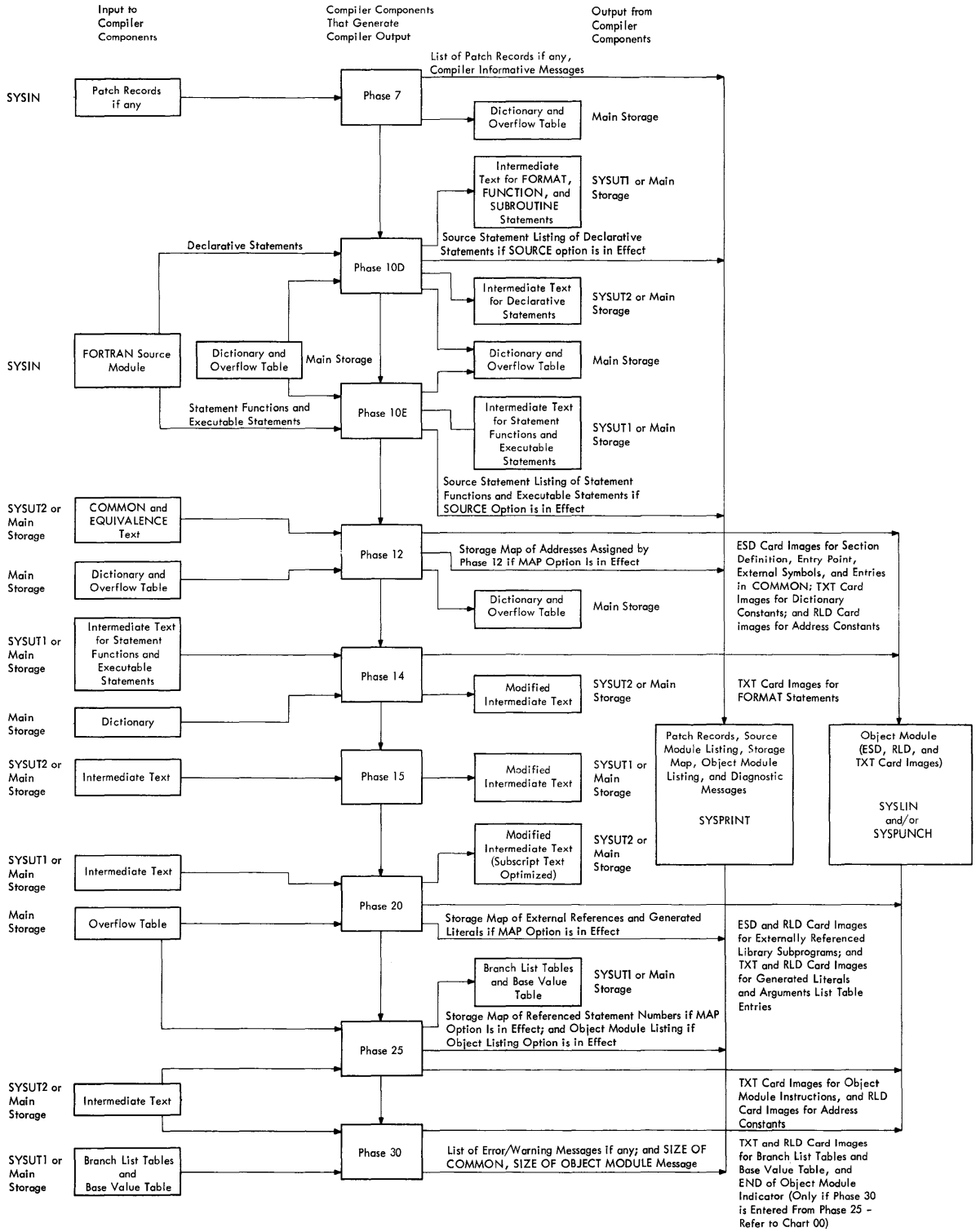


Figure 2. Compiler Input/Output Flow

COMPILER OUTPUT -- THE OBJECT MODULE

The object module compiled from the FORTRAN source module is not constructed in its entirety by any one phase; the various components of the object module are generated throughout the compilation. Figure 3 indicates what each phase contributes to the generation of the object module. An object module is created for use as input to the linkage editor, which prepares object modules for execution on the IBM System/360.

An object module consists of control dictionaries (external symbol dictionary and relocation dictionary), text, and an END statement. The external symbol dictionary (ESD) contains the external symbols that are defined or referred to in the module. The relocation dictionary (RLD) contains information about address constants in the object module. (An address constant designates the relative storage address into which the address of a routine, library subprogram, or symbol is to be relocated.) The text (TXT) contains the instructions and data of the object module. The END statement indicates the end of the object module.

The object module, after being processed by the linkage editor, is executed on the IBM System/360 in conjunction with the following members of the FORTRAN system library (SYS1.FORTLIB):

- IHCFCOME
- IHCFIOSH

IHCFCOME performs object-time implementation of the following FORTRAN statements:

- READ and WRITE
- BACKSPACE, REWIND, and ENDFILE
- STOP and PAUSE

In addition, IHCFCOME converts input and output data into the formats indicated by the FORMAT statements. IHCFCOME also processes object-time errors and arithmetic-type program interruptions and terminates the execution of the load module.

IHCFCOME itself does not actually perform the reading from and writing onto data sets, or I/O device manipulations; it submits requests for such operations to IHCFIOSH (the FORTRAN Input/Output System). IHCFIOSH interprets these requests and submits them to the appropriate BSAM routines for execution.

COMPILER COMPONENTS

The components of the compiler and their main functions are discussed in the following paragraphs.

PHASE 1

Phase 1 is both the first and last phase of the compiler. Initially, the phase is entered from the calling program (e.g., initiator/terminator); subsequent entries are made from either Phase 7 or Phase 30. In addition, if a permanent I/O error occurs, Phase 1 is entered from the phase that requested the I/O operation.

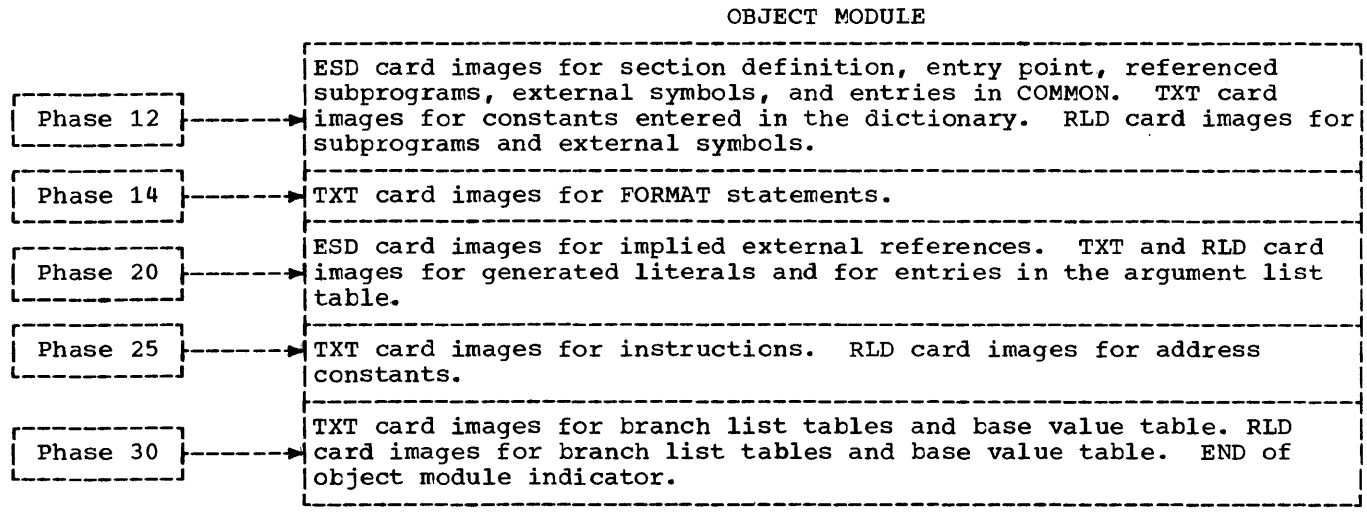


Figure 3. Creation of Object Module

At the initial entry, Phase 1 performs compiler initialization; that is, it loads the interface module and the print buffer module into main storage; processes compiler options; loads the PRFRM module into main storage if the PRFRM option is specified and if the value specified in the SIZE option is at least 17504; opens required data control blocks; and loads Phase 7 into main storage. Upon completion of the initial Phase 1 processing, control is passed to Phase 7.

At subsequent entries, Phase 1 initiates a new compilation if another source module exists, or alternatively terminates the compilation if no more input is present. Control is passed to Phase 7 or returned to the calling program, as appropriate.

INTERFACE MODULE

The interface module contains:

- The communication area (required for compiler communication).
- The data control blocks and data event control blocks for the data sets used during a compilation (required for I/O operations).
- The interface routines (required for implementation of compiler I/O requests and end-of-phase requests, and for temporary modification of compiler components).

PRINT BUFFER MODULE

The print buffer module contains two I/O buffers that are used for the SYSIN and SYSPRINT data sets.

PERFORMANCE MODULE

The performance module, loaded into main storage only if the PRFRM option is specified, contains:

- An I/O routine, which deblocks compiler input and blocks compiler output for PRFRM compilations.
- An end-of-phase routine, which controls the transferring of control from one component of the compiler to the next for PRFRM compilations.
- The blocking table, which provides the I/O routine with the information necessary to deblock compiler input and to block compiler output.
- The BLDL table, which provides the end-of-phase routine with the information necessary to transfer control from one component of the compiler to the next.

PHASE 7

For both SPACE and PRFRM compilations, Phase 7 obtains and allocates main storage for the dictionary, the overflow table, and four internal text buffers. For PRFRM compilations, Phase 7 also obtains and allocates main storage for special I/O buffers to be used for deblocking compiler input and for blocking compiler output if blocking is specified by the user.

After main storage is obtained and allocated, Phase 7 constructs the resident tables to be used by the compiler.

Upon completion of Phase 7 processing control is passed either to Phase 10D or to Phase 1.

PHASE 10D

Phase 10D converts COMMON and EQUIVALENCE source statements into a special form of intermediate text (referred to as COMMON and EQUIVALENCE intermediate text) for processing by Phase 12. In addition, Phase 10D prepares intermediate text and creates dictionary and overflow table entries for specification, FORMAT, SUBROUTINE, and FUNCTION statements for use as input to subsequent phases of the compiler. If the SOURCE option is specified, Phase 10D prepares a list of the statements it processes and writes them on the SYSPRINT data set.

Upon completion of Phase 10D processing, control is passed to Phase 10E.

PHASE 10E

Phase 10E converts statement function definitions, executable statements, and any FORMAT statements interspersed within those statements into intermediate text, which is used as input to subsequent phases of the compiler. During the processing of the above statements, entries are made into the

dictionary and overflow table for the variables, statement numbers, etc., encountered in the statements. If the SOURCE option is specified, Phase 10E also prepares a list of the statements it encounters and writes them on the SYSPRINT data set immediately following the list prepared by Phase 10D.

Upon completion of Phase 10E processing, control is passed either to Interlude 10E for SPACE compilations, or to Phase 12 for PRFRM compilations.

INTERLUDE 10E

Interlude 10E closes and then opens the appropriate data control blocks so that only the BSAM input/output routines required by Phases 12 and 14 are present in as compact an area of main storage as possible. These routines were not called in earlier because storage for them did not exist during the execution of Phases 10D and 10E.

Upon completion of Interlude 10E processing, control is passed to Phase 12.

PHASE 12

Phase 12 assigns relative addresses to symbols entered in the dictionary, overflow table, and COMMON and EQUIVALENCE text. The addresses assigned at this time indicate the relative addresses at which the various symbols will reside in main storage during execution of the load module (i.e., the object module after it has been processed by the linkage editor). Phase 12 also allocates storage for a branch list table for referenced statement numbers and assigns a relative number to each referenced statement number it encounters. Phase 12 generates and then writes ESD and RLD card images for referenced subprograms, and TXT card images for literals on the SYSLIN data set if the LOAD option is specified and/or the SYSPUNCH data set if the DECK option is specified. In addition, if the MAP option is specified, Phase 12 produces a storage map on the SYSPRINT data set of all symbols and literals and their relative addresses.

Upon completion of Phase 12 processing, control is passed to Phase 14.

PHASE 14

Phase 14 reads the intermediate text created by Phases 10D and 10E and replaces any pointers to dictionary entries with information obtained from the dictionary (e.g., with addresses assigned to variables by Phase 12). Phase 14 also converts intermediate text for FORMAT statements into an internal code that is used, at object time, by IHCFCOME, a member of the FORTRAN system library (SYS1.FORTLIB), to place input/output records into the specified formats.

TXT card images for FORMAT statements are generated and then written on the SYSLIN data set if the LOAD option is specified and/or the SYSPUNCH data set if the DECK option is specified. In addition, Phase 14 assigns a position in a second branch list table for each statement function (SF) expansion and DO statement encountered. For SPACE compilations, Phase 14 also frees the storage occupied by the dictionary. (The dictionary is no longer needed after Phase 14 processing.)

Upon completion of Phase 14 processing, control is passed either to Interlude 14 for SPACE compilations, or to Phase 12 for PRFRM compilations.

INTERLUDE 14

Interlude 14, by closing and then opening the appropriate data control blocks, reduces the size of the currently required BSAM input/output routines. This reduction, as well as the freeing of the dictionary area of storage by Phase 14, provides the additional main storage that may be needed for subsequent processing.

Upon completion of Interlude 14 processing, control is passed to Phase 15.

PHASE 15

Phase 15 primarily translates arithmetic expressions into approximate machine code; that is, it produces the data necessary to allow text words to be translated into machine instructions by Phase 25.

Upon completion of Phase 15 processing, control is passed either to Interlude 15 for SPACE compilations, or to Phase 20 for PRFRM compilations.

INTERLUDE 15

Interlude 15, by closing and then opening the appropriate data control blocks, calls in the BSAM input/output routines necessary for performing the I/O operations for the remainder of the compilation.

Upon completion of Interlude 15 processing, control is passed to Phase 20.

PHASE 20

Phase 20 increases the efficiency of the object coding by decreasing the amount of computation associated with subscript expressions. Phase 20 also creates an argument list table to be used, at object time, to provide the addresses of argument lists to subprograms and SFs referenced by the source module. Generated are: ESD card images for any implicitly called library subprograms (refer to the publication IBM System/360 Operating System: FORTRAN IV (E) Library Subprograms); and RLD and TXT card images for any literals generated by the phase and for each entry in the argument list table. These are then written on the SYSLIN data set if the LOAD option is specified and/or the SYSPUNCH data set if the DECK option is specified. In addition, if the MAP option is specified, Phase 20 produces a storage map of the above generated literals and references on the SYSPRINT data set.

Upon completion of Phase 20 processing, if the NOLOAD option is specified and source statement errors were detected, control is passed to Phase 30 to generate error/warning messages; otherwise, control is passed to Phase 25.

PHASE 25

Phase 25 analyzes the text produced by the preceding phases of the compiler and transforms that text into machine language instructions; these instructions become suitable for execution on the IBM System/360 after being processed by the linkage editor. The instructions are generated and written on the SYSLIN data set if the LOAD option is specified and/or the SYSPUNCH data set if the DECK option is specified. Phase 25 completes the assembly of several tables (branch list table for statement numbers, branch list table for SF expansions and DO statements, and a base value table) required for the execution of the instructions generated by the phase. In addition, if the MAP option is specified, Phase 25 produces a list of the referenced statement numbers on the SYSPRINT data set.

Upon completion of Phase 25 processing, control is passed to Phase 30 to generate error/warning messages if necessary, and to process the END statement.

SOURCE SYMBOL MODULE

The source symbol module is used to contain the names of all the variables and constants used in the source module and the relative addresses assigned to them by Phase 12. Phase 25 uses the source symbol module to produce an object module listing if the user specifies the object listing option ($) and if the object listing facility is enabled.

OBJECT LISTING MODULE

The object listing module is loaded into main storage by Phase 25. It is used by Phase 25 to generate the object module listing, if the user specifies the object listing option ($) and if the object listing facility is enabled.

PHASE 30

Phase 30 may be entered from Phase 20 or from Phase 25. When Phase 30 is entered from Phase 20 or Phase 25, any error/warning messages are generated by examining the output text of the preceding phases. Phase 30 also lists the size of COMMON (in bytes), and the size of the object module (in bytes) on the SYSPRINT data set. In addition, if Phase 30 is entered from Phase 25, Phase 30 processes the END statement. This entails generating and writing TXT and RLD card images for the branch list tables, the base value table, and the END card image on the SYSLIN data set if the LOAD option is specified and/or the SYSPUNCH data set if the DECK option is specified.

Upon completion of Phase 30 processing, control is passed to Phase 1.

Section 2 describes the logic and functions of each phase of the compiler.

PHASE 1 (IEJFAAA0/IEJFAAB0)

Phase 1 is both the first and last phase to be executed for each compilation. The phase is initially entered from a calling program (e.g., the initiator/terminator); subsequent entries are made from either Phase 7 if a PFRFM compilation is altered to a SPACE compilation (restart condition), or from Phase 30 -- the last processing phase of the compiler.

At the initial entry (IEJFAAA0), Phase 1 initiates the first compilation and then transfers control to Phase 7.

At subsequent entries (IEJFAAB0), Phase 1 either initiates the next compilation if other source modules are to be compiled, or terminates the compilation (i.e., if no more source modules are present). If a new compilation is initiated, Phase 1 transfers control to Phase 7; if the compilation is terminated, Phase 1 returns control to the calling program.

Chart 01 illustrates the overall logic and the relationship among the routines used in Phase 1. Table 2, the routine directory, lists the routines used in the phase and their functions.

INITIAL ENTRY

At the initial entry, Phase 1 initiates the first compilation. This entails:

- Loading the interface module.
- Loading the print buffer module.
- Processing compiler options.
- Loading the performance module if the PRFRM option is specified and if the value specified in the SIZE option is at least 17504.
- Opening required data control blocks.
- Loading Phase 7.

Loading the Interface Module

When Phase 1 receives control from the calling program, it loads the interface module (IEJFAGA0) into main storage via the LOAD macro-instruction. The interface module contains:

- The communication area.
- DCBs (data control blocks) and DECBs (data event control blocks).
- Interface routines.

COMMUNICATION AREA: The communication area contains information that must be communicated between the various components of the compiler. The communication area contains the following type of information:

- User-specified information, that is, options and parameters chosen by the user to tailor the output of a compilation to his specifications (e.g., DECK).
- Default values for compiler options. The interface module is assembled, and processed by the linkage editor during system generation. This allows the user to specify default values for compiler options (refer to the publication IBM System/360 Operating System: System Generation). These default values will be assumed if the corresponding values in the PARM field of the EXEC statement for a FORTRAN compilation are not included by the user. (Refer to Appendix K for the default values that may be specified during the system generation process.)
- Information required for communication between the compiler and the operating system, such as:
  1. Branch instructions to specific routines in the interface module. (For PRFRM compilations, these branch instructions are, in effect, replaced by branch instructions to routines in the performance module.)
  2. A pointer to DCBs (data control blocks) and the DECBs (data event control blocks) needed for input/output operations during the compilation.

- Compilation information, such as:

  1. Type of program/subprogram being compiled (i.e., main program, FUNCTION subprogram, or SUBROUTINE subprogram).

  2. Size of internal text buffers.

  3. Addresses of buffers, table indexes, certain tables, and work areas.

  4. Indicators (e.g., indicators of any errors encountered during the compilation).

- Object-time information, such as:

  1. Size of COMMON to be used with the object module, and of the tables required for the object module execution.

  2. The location counter used, throughout the compilation, for the assignment of object-time addresses.

DCBS AND DECBS: The DCBs and DECBs for the data sets used during the compilation are assembled into the interface module in skeletal form. (For a description of the DCBs and DECBs refer to the publication IBM System/360 Operating System: Introduction to Control Program Logic, Program Logic Manual.) The various fields of the DCBs are filled in by the control program when the data control blocks are opened (refer to the publication IBM System/360 Operating System: Concepts and Facilities). However, the DCB block size fields for data sets SYSUT1 and SYSUT2 are overlayed with values computed by the compiler.

INTERFACE ROUTINES: The interface module contains four interface routines: an I/O routine, an end-of-phase routine, a print control operations routine, and a patch routine. (See Chart 02).

The I/O routine (SIORTN) processes I/O requests of the compiler. For SPACE compilations, the I/O requests are initiated via a linkage to this routine. (Refer to Appendix H for a description of this linkage to the interface module.) For PRFRM compilations, the I/O requests are initiated via a linkage to the PIORTN routine in the performance module. The PIORTN, in turn, links to the SIORTN routine in the interface module. The SIORTN routine:

- Analyzes the linkage parameters passed to it by either the component of the compiler requesting I/O, or other interface module routines. These parameters indicate: (1) the type of request (read, write, or check), (2) the address of the I/O buffer for the operation, and (3) what data set is to be used for the operation.

- Fulfills the request by issuing the appropriate macro-instruction (READ, WRITE, and/or CHECK).

The compile-time I/O error recovery procedure is illustrated in Chart 02.

The end-of-phase routine (SNEXT) is the means by which control is passed from one component of the compiler to the next for SPACE compilations. The transferring of control between compiler components is initiated via a linkage to this routine. (Refer to Appendix H for a description of this linkage to the interface module.) The end-of-phase routine:

- Analyzes the linkage parameters passed to it by the component of the compiler relinquishing control. These parameters indicate the name of the next component to be executed and the disposition of various data sets.

- Repositions the data sets indicated in the linkage parameters.

- Transfers control to the next component via the XCTL macro-instruction.

The print control operations (PRTCTRL) routine allows the use of device-independent control operations for the SYSPRINT data set. If the data set is being placed onto an intermediate storage device before being printed, the printer control codes remain as part of the data set (thereby retaining device independence).

The patch routine (PATCH) allows temporary modification of the compiler modules. (A module is modified for the duration of a batch compilation.) Each compiler module unconditionally branches to the patch routine to check whether the module being executed is to be modified. (Refer to Appendix H for a description of this linkage to the interface module.) If it is, the patch routine overlays the instructions or data of the module to be modified with patch information for that module. This information is placed in the patch table (a 100-byte portion of the patch routine) by Phase 7. If there is no patch information, control is immediately returned to the module being executed.

Loading the Print Buffer Module

The print buffer module (IEJFAKA0) is loaded into main storage during Phase 1. It contains two I/O buffers that are used by the SYSIN and SYSPRINT data sets. SYSIN uses the I/O buffers during the source statement scan. The card images of the source module(s) to be compiled are alternately read into one of the two buffers. The double-buffer scheme allows for overlapping the scanning of a card image in one buffer with the reading of the next card image of the source module into the other buffer.

SYSPRINT uses the I/O buffers for: (1) writing patch records if any, (2) generating the storage map, and (3) listing the source module.

Processing Compiler Options

Options may be chosen by the user to tailor the output of the compiler to his specifications. Phase 1 checks these options specified in the execute statement (EXEC) for the compilation. This information was previously entered into an area designated by the calling program. The contents of this area are obtained by Phase 1 via an address in general register 1. They are then encoded and entered in the communication area. For a description of the options and their use, refer to the publication IBM System/360 Operating System: FORTRAN IV (E) Programmer's Guide.

If the object listing facility of the compiler has been enabled, Phase 1 also checks whether the object listing option (a $ in the PARM field of the EXEC statement) is specified. (The object listing facility is enabled by reassembling Phase 1 with the branch instruction that disables the facility either removed or replaced with a no-op instruction.) If the option is specified, Phase 1: (1) sets the appropriate indicator in the communication area, and (2) loads the source symbol load module (SORSYM) into main storage. SORSYM, a SYS1.LINKLIB load module (IEJFAXA0), reserves an area in main storage. The names of all variables and constants used in the source module and their corresponding relative addresses are placed into this area by Phase 12.

If the object listing facility has not been enabled, Phase 1 indicates an invalid compiler option, by setting the invalid option bit in the communication area, if the object listing option is specified.

Loading the Performance Module

Phase 1 examines the PRFRM bit in the communication area to determine if the PRFRM option has been specified by the user. If the PRFRM option is specified, and if the value specified in the SIZE option is at least 17504, Phase 1 loads the performance module (IEJFAPA0) into main storage. The performance module reduces phase-to-phase transition processing and thereby decreases compilation time. The performance module is composed of two routines and two tables.

PERFORMANCE MODULE ROUTINES: The performance module contains an I/O routine, and an end-of-phase routine. (See Chart 03.)

The I/O routine (PIORTN) is used to deblock compiler input on SYSIN; and to block compiler output on SYSLIN, SYSPRINT, and SYSPUNCH, as required by the block sizes specified for the above data sets. I/O requests for a PRFRM compilation are initiated via a linkage to this routine. (Refer to Appendix H for a description of this linkage to the performance module.) The I/O routine:

- Analyzes the linkage parameters passed to it by the calling phase. These parameters indicate: (1) the type of request (read, write, check, or flush), (2) the address of the area into which, or from which the logical record is to be moved, and (3) the data set to be used for the operation. (A flush request forces the contents of the current output buffer to be written out.)

- Deblocks compiler input from SYSIN if a blocking factor greater than 1 is specified. The PIORTN routine reads (via a linkage to the SIORTN routine in the interface module) a block from the SYSIN data set into an I/O buffer only when an entire block has been deblocked and moved into the area requested by the calling phase. This reduces the number of READ macro-instructions issued for a compilation and thus decreases compilation time.

- Blocks compiler output on the output data sets if their corresponding blocking factors are greater than 1. (Each blocking factor is determined from the BLKSIZE (block size) field in the DCB parameter of the associated DD statement.) In general, the PIORTN writes (via a linkage to the SIORTN routine in

the interface module) a block onto an output data set only when the I/O buffer containing that block has been filled. (However, when Phase 1 requests a flush at the end of the last compilation, the PIORTN will force a truncated buffer to be written if the buffer is only partially filled.) This reduces the number of WRITE macro-instructions issued for a compilation and thus decreases compilation time.

The end-of-phase routine (PNEXT) is the means by which control is passed from one component of the compiler to the next for PRFRM compilations. The transferring of control between compiler components is initiated via a linkage to this routine. (Refer to Appendix H for a description of this linkage to the performance module.) The end-of-phase routine:

- Analyzes the linkage parameters passed to it by the component of the compiler relinquishing control. These parameters indicate the name of the next component to be executed, and the disposition of the various data sets.
- Repositions the data sets indicated in the linkage parameters.
- Transfers control to the next component via the XCTL macro-instruction. If the next component is an interlude, the performance module bypasses the execution of the interlude and transfers control to the next phase of the compiler. If the next component is a phase, the performance module immediately transfers control to the next phase.

PERFORMANCE MODULE TABLES: The performance module contains two tables: the blocking table, and the BLDL table.

Phase 7 constructs a blocking table entry for each of the data control blocks that are opened by Phase 1. The blocking table provides the PIORTN routine with the information necessary to deblock compiler input, and to block compiler output. (Refer to Appendix C for the format of the blocking table.)

Phase 7 constructs the BLDL table via the BLDL macro-instruction. The BLDL table provides the PNEXT routine with the information necessary to transfer control from one component of the compiler to the next. (Refer to Appendix C for the format of the BLDL table.)

Opening Required Data Control Blocks

The data control blocks that are opened by Phase 1 depends upon the options specified by the user.

If the SPACE option is specified, Phase 1 opens (via the OPEN macro-instruction) only the data control blocks for the data sets used by Phases 7, 10D, and 10E (SYSIN, SYSUT1, SYSUT2, and SYSPRINT). The main storage that is saved at this time by not opening the data control blocks for SYSLIN and SYSPUNCH is necessary for the execution of Phases 10D and 10E. (The SYSLIN and SYSPUNCH data sets are not needed by the compiler until the execution of Phase 12. Therefore, their corresponding data control blocks are not opened until the execution of Interlude 10E.)

If the PRFRM option is specified, Phase 1 opens (via the OPEN macro-instruction) the data control blocks for all the data sets required by the compiler. Because all the required data control blocks are opened initially, the compiler can bypass the execution of Interludes 10E, 14, and 15; and can avoid repeated closing and reopening of data control blocks. Bypassing the execution of the interludes reduces phase-to-phase transition time and thus decreases compilation time.

If neither the SPACE nor the PRFRM option is specified, Phase 1 assumes a default value of SPACE and opens the data control blocks accordingly.

The manipulation of data control blocks by subsequent components of the compiler for SPACE compilations as well as for PRFRM compilations is illustrated in Appendix A.

Loading Phase 7

Phase 7 (IEJFEAA0) is loaded into main storage by Phase 1, using the LOAD macro-instruction. This is not the normal condition; normally, the XCTL macro-instruction in the end-of-phase routine is used to call a phase into main storage.

Phase 1 loads Phase 7 into the highest area of available main storage, relative to location zero. (The XCTL macro-instruction would load Phase 7 into the lowest area of available main storage.) This special loading by Phase 1 permits Phase 7 to set up the resident tables in the lowest area of available main storage. The physical locations occupied by the various compiler components and resident tables are illustrated in Appendix J.

SUBSEQUENT ENTRIES

At subsequent entries, Phase 1 either:

- Initiates a new compilation, or
- Terminates the compilation.

Initiating a New Compilation

If a new compilation is to be initiated, Phase 1 first determines if a PRFRM or a SPACE compilation is to be performed. If a PRFRM compilation is to be performed, Phase 1 immediately loads (via the LOAD macro-instruction) Phase 7 into main storage and then transfers control to Phase 7.

If a SPACE compilation is to be performed, Phase 1 determines if a restart condition exists. That is, if a PRFRM compilation was requested and Phase 7 determined that the required main storage for the PRFRM compilation was not available. Phase 7 then alters the PRFRM compilation to a SPACE compilation and returns control to Phase 1.

If a restart condition exists, Phase 1: (1) deletes (via the DELETE macro-instruction) the performance module from main storage, (2) closes (via the CLOSE macro-instruction) the data control blocks for all required compiler data sets (opened by Phase 1 for the PRFRM option), and (3) reopens (via the OPEN macro-instruction) only the data control blocks for the data sets required for Phases 7, 10D, and 10E. Phase 1 then loads (via the LOAD macro-instruction) Phase 7 into main storage and transfers control to Phase 7.

If a restart condition does not exist and if the SPACE option is in effect, Phase 1 first frees (via the FREEMAIN macro-instruction) the main storage that was previously allocated to the compiler for the internal text buffers and the overflow table during execution of Phase 7. Subsequent Phase 1 processing except for the deletion of the performance module is the same as that described for the restart condition.

Terminating the Compilation

If the last source module on the SYSIN data set has been compiled, Phase 1 first requests a flush operation for the SYSLIN, SYSPUNCH, and SYSPRINT data sets. A flush request forces the current output buffer being used for a blocked data set to be written. This insures that all compiler output for blocked data sets is written. In the case of an unblocked data set, the flush request for that data set is ignored. Phase 1 next closes (via the CLOSE macro-instruction) the data control blocks for all the data sets used by the compiler. Phase 1 then: (1) frees (via the FREEMAIN macro-instruction) all the main storage that was allocated to the compiler during execution of Phase 7, and (2) deletes (via the DELETE macro-instruction) the interface module, the print buffer module, and, for a PRFRM compilation, the performance module. Control is then returned to the calling program with the proper return code.

If internal errors (e.g., permanent I/O errors) occur at any time, the current compilation is immediately terminated by calling Phase 1. Phase 1 then performs the above processing and returns control to the calling program with a return code of 16.

PHASE 7 (IEJFEAA0)

Phase 7, the second phase of the compiler, is entered after the completion of Phase 1. The functions of the phase are:

- Obtaining main storage for the compiler.
- Allocating main storage to the compiler.
- Constructing resident tables used by the compiler.

At the conclusion of Phase 7 processing, a delete routine is moved into the print buffer module. Control is then passed to the delete routine. The delete routine deletes Phase 7 from main storage (via the DELETE macro-instruction) and then passes control to either Phase 1 (to restart or terminate a compilation) or to Phase 10D (to begin the scan of source module statements).

Chart 04 illustrates: (1) the overall logic and the relationship among the routines of Phase 7, and (2) the overall logic of the delete routine. Table 3, the routine directory, lists the routines used in the phase and their functions.

OBTAINING MAIN STORAGE

The amount of main storage required by the compiler depends on whether a SPACE or a PRFRM compilation is being performed. For a SPACE compilation, a minimum of

15,360 bytes is required. For a PRFRM compilation, a minimum of approximately 19,500 bytes is required. (The exact amount depends on the device configuration of the user. That is, different I/O devices require different access method routines and different control blocks.)

The process of obtaining main storage is actually started in Phase 1. Phase 1 has already obtained main storage for:

- The interface module.
- The print buffer module.
- The performance module (if the PRFRM option is specified).
- BSAM routines.
- Phase 7.

Phase 7, upon receiving control from Phase 1, calculates the total amount of main storage obtained by Phase 1, and subtracts this amount from the value specified in the SIZE option. (If the SIZE option was not specified by the user, the minimum amount required for a SPACE compilation is assumed as a default value for the SIZE option.) The result of this calculation is the amount of main storage that Phase 7 attempts to obtain via the GETMAIN macro-instruction. If more than this amount is obtained, Phase 7 frees the excess via the FREEMAIN macro-instruction. If less than the minimum amount required for a SPACE compilation is obtained, an unconditional GETMAIN macro-instruction is issued in order to obtain the minimum amount.

ALLOCATING MAIN STORAGE

The procedure used by Phase 7 for allocating main storage depends on whether a SPACE or a PRFRM compilation has been initiated. Appendix J illustrates the main storage allocated to the compiler for both SPACE and PRFRM compilations.

For SPACE Compilations

For a SPACE compilation, the main storage obtained by Phase 7 is allocated, via the storage allocation table, among the transient work area (an 800-byte area required by the control program), the dictionary, the overflow table, and four internal text buffers. The storage allocation table (refer to Appendix B) indicates the amount of main storage to be allocated to the text buffers, the dictionary, and the overflow table.

The main storage allocated to the dictionary and the overflow table, except for the reserved word portion of the dictionary, may be segmented. That is, the dictionary and overflow table may occupy more than one segment of main storage. The location of the segments allocated to the dictionary and overflow table are recorded (sequentially by address) in a segment address list (SEGMAL). SEGMAL resides at the beginning of the first segment. The location of the dictionary index and the overflow table index as well as a pointer to the ending location of the current segment in which the dictionary and overflow table are being built are recorded in the communication area.

The dictionary portions are loaded into the highest storage segment(s) and the overflow table portions are loaded into the lowest storage segment(s). This ensures that the dictionary resides "above" the overflow table. The dictionary must reside above the overflow table because the storage allocated to the dictionary is freed (via the FREEMAIN macro-instruction) at the conclusion of Phase 14 processing. This additional main storage is required for the execution of subsequent phases, primarily for Phase 15. (For PRFRM compilations, the main storage allocated to the dictionary is not freed until compilation is terminated by Phase 1.)

The main storage allocated to the internal text buffers may be segmented. However, the main storage for each buffer itself must be contiguous. The location of the segment assigned to each buffer is indicated in the communication area.

For PRFRM Compilations

For a PRFRM compilation, the main storage allocation algorithm must determine if blocked I/O is specified by the user.

BLOCKED I/O: If any blocked I/O is specified, portions of the obtained main storage must be allocated to special I/O buffers required for blocking and deblocking. Phase 7 allocates main storage for two I/O buffers for each data set for which blocking is requested. The size of each buffer is determined by the BLKSIZE field in the DCB parameter of the associated DD statement. If the BLKSIZE fields are not specified, the compiler assumes the following default values for the compiler data sets:

- SYSPRINT -- 121
- All others -- 80

After allocating main storage for the special I/O buffers, Phase 7 determines if sufficient storage remains for the transient work area, the dictionary, the overflow table, and the four internal text buffers. If there is sufficient storage, subsequent main storage allocation for a PRFRM compilation with blocked I/O is the same as that described for a SPACE compilation.

In the event that the remaining main storage is not sufficient, the compilation is terminated and control is transferred to Phase 1. Phase 1, in turn, passes control to the scheduler to terminate the job step.

UNBLOCKED I/O: If all I/O is unblocked, Phase 7 determines if the amount of main storage obtained is sufficient for the transient work area, the dictionary, the overflow table, and the internal text buffers. If there is sufficient storage, subsequent main storage allocation for a PRFRM compilation with unblocked I/O is the same as that described for a SPACE compilation.

If the amount of main storage obtained is not sufficient, Phase 7 frees (via the FREEMAIN macro-instruction) all the main storage it obtained. Phase 7 then alters the PRFRM compilation to a SPACE compilation (restart condition) and transfers control to Phase 1 via the delete routine. Phase 1 then initializes the compiler for a SPACE compilation.

RESIDENT TABLE CONSTRUCTION

The resident tables of the compiler (described in Appendix C) are:

- The dictionary and the overflow table.
- The segment address list (SEGMAL).
- The patch table.
- The blocking table and the BLDL table (resident only for PRFRM compilations).

For the dictionary and the overflow table, Phase 7 only constructs the portions that are independent of the source module being compiled. SEGMAL is constructed as main storage segments are allocated to the dictionary and the overflow table. The patch table, a portion of the interface module, is constructed only if the patch facility has been enabled and if patch records precede the source statements of the source module(s) being compiled. The blocking table and the BLDL table, portions of the performance module, are constructed only for PRFRM compilations.

Dictionary and Overflow Table

Phase 7 constructs only those portions of the dictionary and overflow table that are independent of the source module being compiled. In the dictionary, the index and the reserved word portion are constructed. In the overflow table, the overflow index is constructed.

The index for the dictionary and the index for the overflow table are used by subsequent phases to enter information into and obtain information from the respective table. The reserved word portion of the dictionary contains all the reserved words of the FORTRAN IV (E) language.

SEGMAL

SEGMAL contains the starting and ending addresses of each main storage segment allocated to the dictionary and the overflow table. The starting address and the length of each segment is obtained as a result of the GETMAIN macro-instruction. Phase 7 then computes the ending address of each segment, and enters both the starting and ending address for each segment into SEGMAL. This sequence of addresses constitutes SEGMAL.

Patch Table

If the patch facility of the compiler has been enabled, Phase 7 determines if the first record read from SYSIN is a patch record. (The patch facility is enabled by reassembling Phase 7 with the branch instruction that disables the patch facility either removed or replaced with a no-op instruction.) If the first record is a patch record, it is first listed on SYSPRINT and then posted in a patch table (100 bytes) in the interface module. Posting consists of: (1) converting the contents of a patch record into a format that is usable to the patch routine, and (2) moving the converted patch record to the patch table. All subsequent patch records are processed in this manner by Phase 7.

Blocking Table and BLDL Table

Phase 7 constructs the blocking table and the BLDL table only for PRFRM compilations. The performance module contains the main storage required for these tables.

Phase 7 constructs a blocking table entry for each of the data control blocks that were opened by Phase 1. Phase 7 places information into the blocking table that is required for deblocking compiler input and blocking compiler output. This information includes such things as: logical record length, blocking factor, pointers to the special buffers allocated by Phase 7, etc.

Phase 7 constructs the BLDL table via the BLDL macro-instruction. (For a description of the BLDL macro-instruction, refer to the publication IBM System/360 Operating System: Data Management.) The BLDL table contains the information necessary to transfer control from one component of the compiler to the next. The construction of the BLDL table reduces phase-to-phase transition time and thereby decreases compilation time.

PHASE 10D (IEJFGAA0)

Phase 10D, the first processing phase of the compiler, is entered after the completion of Phase 7. This phase processes the specification statements of the source module (plus the FUNCTION or SUBROUTINE statement if a subprogram is being compiled). These statements, which are called declarative statements, are:

- COMMON
- DIMENSION
- EQUIVALENCE
- INTEGER
- REAL
- DOUBLE PRECISION
- EXTERNAL
- FORMAT
- SUBROUTINE or FUNCTION

Declarative statements, other than the FORMAT statement, must precede the statement function definitions and the executable statements. The executable statements are all FORTRAN IV (E) statements other than those listed above and statement function definitions.

In processing the declarative statements, Phase 10D performs the following functions:

- Prepares intermediate text.
- Constructs dictionary and overflow table entries.
- Prepares the first part of the source statement listing (a minor function).

Phase 10D and Phase 10E (the next phase to be executed) convert each FORTRAN source statement into usable input to subsequent phases of the compiler. Phase 10D converts the declarative statements; Phase 10E converts the statement function definitions and the executable statements. The result of this conversion is intermediate text (an internal representation of the source statements), and the dictionary and overflow table that contain detailed information about specific portions of the statement.

The information in the dictionary and overflow table supplements the intermediate text in the generation of code by the succeeding phases. This information is associated with the intermediate text entries via pointers that reside in the text entries.

A complete listing of the declarative statements is prepared on the SYSPRINT data set by Phase 10D if the SOURCE option has been chosen.

When a statement function definition or an executable statement is encountered in the input stream, control is passed to Phase 10E.

Figure 4 illustrates the data flow within the phase.

Chart 05 indicates the overall logic and the relationship among the routines of Phase 10D. Table 5, the routine directory, lists the routines used in the phase and their functions.

INTERMEDIATE TEXT PREPARATION

Phase 10D produces intermediate text, which is the form in which information is transmitted from the source module to the processing phases. (Refer to Appendix L for a description of the source statement scan required for intermediate text preparation.)

Intermediate text is prepared for FORMAT, FUNCTION, and SUBROUTINE declarative statements. (Refer to Appendix D for the intermediate text format.) This text is used to transmit these statements to Phases 14, 15, 20, and 25.

Two special forms of intermediate text, COMMON and EQUIVALENCE text, are produced for COMMON and EQUIVALENCE statements, respectively. (Refer to Appendix D for the format.) These special forms of text transmit the corresponding statements to Phase 12.

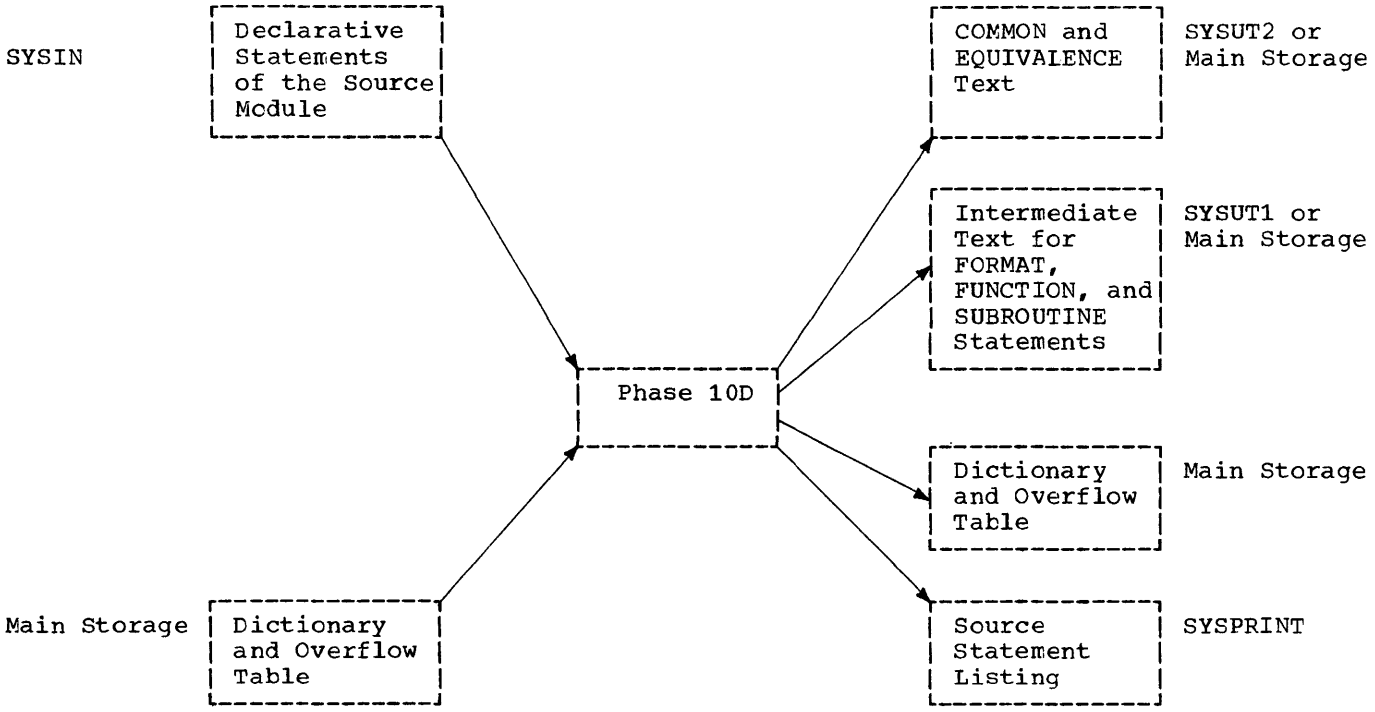


Figure 4. Phase 10D Data Flow

CONSTRUCTION OF DICTIONARY AND OVERFLOW TABLE ENTRIES

Dictionary and overflow table entries are made during Phase 10D for:

- Symbols appearing within declarative statements.
- Statement numbers associated with declarative statements.

Entries are made to the dictionary (refer to Appendix C) for symbols appearing in all declarative statements except the FORMAT statements. If any symbol is already entered in the dictionary, that entry is modified, if necessary, to reflect any new information about the symbol under consideration. For example, if the symbol is in COMMON, an indicator in the dictionary is set on.

Entries are made to the overflow table (refer to Appendix C) for:

- Statement numbers.
- Dimension information.

PHASE 10E (IEJFJAA0)

Phase 10E, the second processing phase of the compiler, is entered after the completion of Phase 10D. The functions of the phase are:

- Intermediate text preparation.
- Construction of dictionary and overflow table entries.
- Completion of the preparation of the source statement listing (a minor function).

Phase 10E processes SFs (statement functions), the executable statements of the source module, and any FORMAT statements interspersed among them. As each SF, executable, or FORMAT statement appears in the input stream, intermediate text is prepared and corresponding entries are made to the resident tables. The intermediate text prepared by Phase 10E represents the executable source module statements. The resident tables complement intermediate text. (For the formats of the intermediate text and the resident tables, refer to Appendixes D and C, respectively.) If any syntactical errors are encountered during

the processing of an SF, executable, or FORMAT statement, error intermediate text entries are made immediately following the intermediate text entries for the statement in which the error was detected.

As the intermediate text is prepared from the source statements processed by Phase 10E, a list of these statements is added to the SYSPRINT data set, which was begun by Phase 10D.

When the END statement is encountered, Phase 10E passes control either to Interlude 10E (IEJFJGA0) for SPACE compilations, or to Phase 12 for PRFRM compilations.

Figure 5 illustrates the data flow within the phase. The data sets SYSIN, SYSUT1, and SYSPRINT are not repositioned after Phase 10D; therefore, Phase 10E can continue to read from SYSIN or to add to SYSUT1 and SYSPRINT.

Chart 06 illustrates the overall logic and the relationship among the routines of Phase 10E. Table 7, the routine directory, lists the routines used in the phase and their functions.

INTERMEDIATE TEXT PREPARATION

Phase 10E produces intermediate text for each SF and executable statement, and for any FORMAT statements among them. (Refer to Appendix L for a description of the source statement scan required for intermediate text preparation.)

For a subscripted expression appearing within a statement, a unique intermediate text entry of two words is made (refer to Appendix D). The offset of the subscripted expression (for which a field in this unique text entry is reserved) is computed by Phase 10E. For a discussion of this aspect of subscripted expressions, refer to Appendix E.

The combination of the intermediate text prepared by Phase 10D and the intermediate text prepared by Phase 10E form the intermediate text that is manipulated in the succeeding phases.

CONSTRUCTION OF DICTIONARY AND OVERFLOW TABLE ENTRIES

Phase 10E makes entries to the dictionary for:

- Variables.
- Constants.
- Subprograms.
- Data set reference numbers.

(Refer to Appendix C for the format and content of these entries.)

Phase 10E makes entries to the overflow table for:

- Subscripted expressions appearing in the executable statements.
- Statement numbers associated with FORMAT statements or executable statements.

(Refer to Appendix C for the format and content of these entries.)

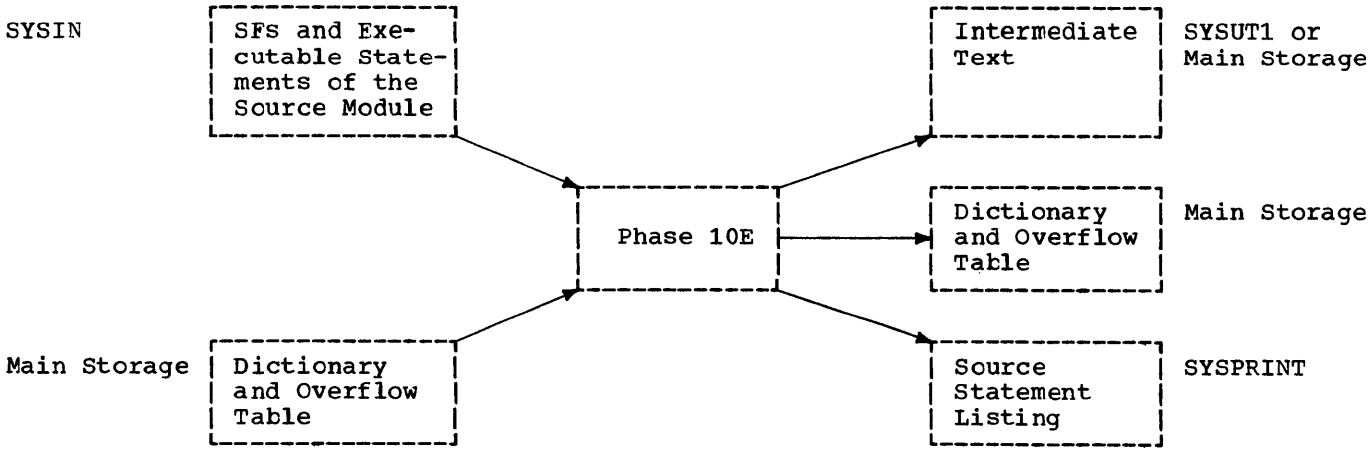


Figure 5. Phase 10E Data Flow

PHASE 12 (IEJFLAA0)

Phase 12, the third processing phase of the compiler, is entered either after the completion of Interlude 10E for SPACE compilations, or after the completion of Phase 10E for PRFRM compilations. The functions of the phase are:

- Address assignment.
- EQUIVALENCE statement processing.
- Branch list table preparation.
- Card image preparation.
- Preparation of a storage map if the MAP option is specified (a minor function).

Address assignment is the allocation of relative storage locations to:

- Variables and arrays in COMMON.
- Equated variables.
- Nonequated variables and arrays in the dictionary (dictionary entries).
- Constants.
- Variables in subscripted expressions.

Addresses are assigned in the order in which they are listed above.

If the object listing facility of the compiler has been enabled and if the object listing option is specified, Phase 12 places the names of all variables and constants used in the source module and their corresponding relative addresses into the SORSYM load module. (SORSYM was previously loaded into main storage by Phase 1.)

Processing of the EQUIVALENCE text occurs after the assignment of addresses to variables and arrays in COMMON but before the assignment of addresses to dictionary entries.

EQUIVALENCE text processing assigns relative positions to the variables within the EQUIVALENCE statements. These relative positions are indicated in a table, which is created and used to assign relative addresses to the variables according to their position in the table.

After the assignment of addresses to variables in subscripted expressions, Phase 12 prepares a branch list table, which is used to control branching within the object module.

During the assignment of addresses by Phase 12, ESD, TXT, and RLD card images are generated for section definitions, entry points, literals, and external references.

In addition to the preceding functions, Phase 12 prepares a storage map to indicate all address assignments made during the phase.

After the completion of Phase 12 processing, control is passed to Phase 14.

Figure 6 illustrates the data flow within the phase.

Chart 07 illustrates the overall logic of Phase 12 and the relationship among its routines. Table 8, the routine directory, lists the routines used in the phase and their functions.

ADDRESS ASSIGNMENT

An effective address in IBM System/360 Operating System (a base-displacement address) is the displacement in an instruction added to the value in a base register. This yields a two-byte address wherein the

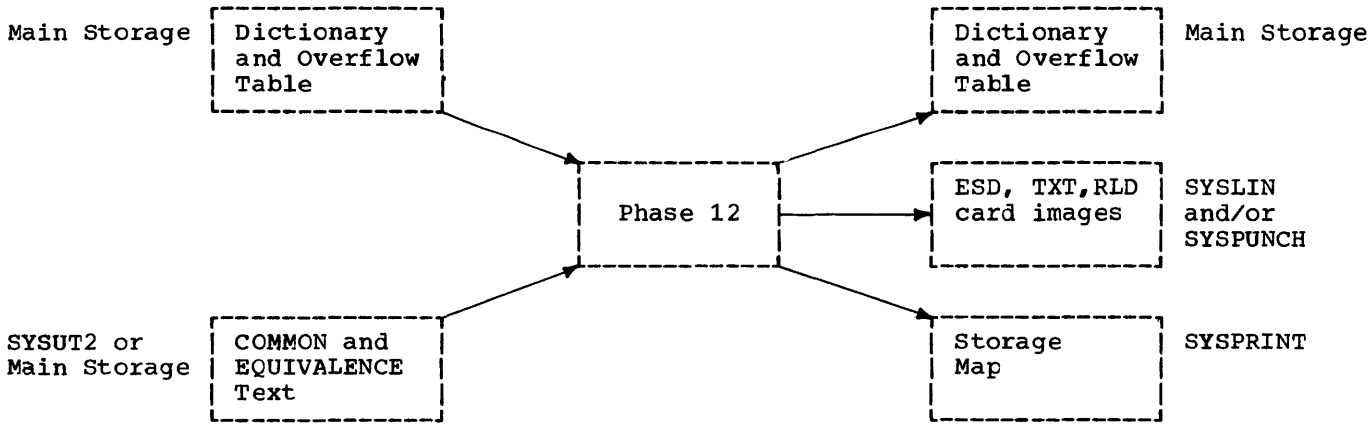


Figure 6. Phase 12 Data Flow

first four bits represent a general register used as a base register and the last twelve bits represent the displacement. All symbols in the object module generated by the compiler are referenced by this two-byte address.

The base-displacement address is assigned through the use of a location counter, which is initialized and then incremented by the number of bytes needed in main storage to contain the variable, array, constant, address constant, or equated variable assigned an address. If more than 4096 bytes are needed, a new base register is assigned.

There are only two instances in which the location counter may be incremented when no address is assigned:

- The first occurs after the variables in COMMON are assigned addresses. A new base register is assigned to the location counter so that a variable in COMMON has a different base register than a variable not in COMMON.
- The second may occur after integer and real constants are assigned addresses. The location counter is adjusted to accommodate the double-precision constants. Double-precision constants are assigned addresses immediately after real and integer constants.

When a variable is assigned an address, that address is placed in the chain address field of the dictionary entry for the variable.

FORMAT statements are assigned addresses during the execution of Phase 14. All phases after Phase 12 assign addresses whenever a constant or work area is defined.

EQUIVALENCE STATEMENT PROCESSING

The EQUIVALENCE text is processed by Phase 12 so that equated variables are assigned to the same address.

The following terms are used in the description of EQUIVALENCE processing:

- EQUIVALENCE group -- the variable and/or array names between a left and right parenthesis in an EQUIVALENCE statement.
- EQUIVALENCE class -- two or more EQUIVALENCE groups that have the following characteristic. If any EQUIVALENCE groups contain the same element, these groups form an EQUIVALENCE class. Further, if any other group contains an element in this class, the other group is part of this class, etc.
- Root -- the member of an EQUIVALENCE group or class from which all other variables in that group or class are referenced by means of a positive displacement.
- Displacement -- the distance, in bytes, between a variable and its root.

The root of an EQUIVALENCE group is assigned an address, and all other variables in the group are assigned addresses relative to that root.

To determine the root and the displacement of the other elements in the group from the root, the first element in the EQUIVALENCE group is established initially as the root. The displacement for the other elements (in relation to the root) is calculated by subtracting the offset of the root from the offset of the variable whose displacement is being calculated. (The offset for subscripted variables is contained in the EQUIVALENCE text created by Phase 10D. The offset for nonsubscripted variables is zero.)

If the resulting displacement is negative, the root is changed. The new root is the variable whose displacement was being calculated. Whenever a new root is assigned to an EQUIVALENCE group, the previously calculated displacements must be recalculated.

The root and the displacements in each group are entered in an EQUIVALENCE table, which is used by the storage assignment routines of Phase 12 to assign addresses to equated variables. (Refer to Appendix B for the table format.)

BRANCH LIST TABLE PREPARATION

The branch list table is initialized by Phase 12 (and is completed by Phase 25). This table is used by the object module to control the branching process. (Refer to Appendix F for the table format.) Each statement number referenced in a control statement is assigned a position relative to the start of the branch table. This position is indicated to Phase 25 by a relative number, which replaces the chain field of the corresponding statement number entry in the overflow table.

In the assignment process, the statement number chains in the overflow table are

scanned sequentially. Each time an entry for a statement number indicates a referenced statement other than the statement number of a FORMAT or specification statement, a counter associated with the branch list table is incremented by 4. (Four bytes are required for the referenced statement number and the address that will be assigned to the number by Phase 25.) The current contents of that counter are then placed in the chain field of the corresponding overflow table entry.

This counter is initialized to 0. Therefore, the first statement number in the first chain is assigned the number 0, the second statement number is assigned the relative number 4, the third statement number is assigned the relative number 8, and so on. After all statement numbers are assigned, the location counter is incremented by an amount equal to the size of the branch list table (in bytes).

CARD IMAGE PREPARATION

Several card images are prepared during the execution of Phase 12. This involves setting up the proper formats for the card images and inserting the pertinent information into those formats. The card images prepared are indicated below, along with their functions. For a more complete discussion of the use and format of these cards, refer to the publication IBM System/360 Operating System: Linkage Editor, Program Logic Manual.

The cards generated by Phase 12 are:

- ESD-0 This is the section definition card for the source module being compiled.
- ESD-1 This card defines the entry point for the source module being compiled.
- ESD-2 This card is produced for external subprogram names. There may be several such cards.
- ESD-5 This is the section definition card for COMMON (if a COMMON statement exists in the source module being compiled).
- TXT This card is produced for constants that have been entered in the dictionary. There may be several such cards.
- RLD This card contains the address of the location at which the address of each external subprogram will be loaded at object time. There may be several such cards.

PHASE 14 (IEJFNAA0)

Phase 14, the fourth processing phase of the compiler, is entered after the completion of Phase 12. The functions of the phase are:

- FORMAT statement processing.
- READ/WRITE statement processing.
- Replacing dictionary pointers.
- Miscellaneous statement processing.

The FORMAT statement processing converts the intermediate text for FORMAT statements into a form acceptable to IHCFCOME and creates TXT card images. These card images are used by IHCFCOME to set up the format of the list items for the I/O operations of the compiled source module. For a discussion of IHCFCOME, refer to Appendix G.

The processing for READ/WRITE statements consists of checking the components of the READ/WRITE statements for validity, processing implied DOs within the READ/WRITE statements, and rearranging the intermediate text for READ/WRITE statements.

Phase 14 replaces dictionary pointers in the intermediate text with the appropriate address assigned by Phase 12, a data set reference number, or a statement function number. (For SPACE compilations, the main storage occupied by the dictionary is freed by Phase 14.)

Upon completion of the Phase 14 processing, control is passed either to Interlude 14 (IEJFNGA0) for SPACE compilations, or to Phase 15 for PRFRM compilations.

The input to Phase 14 is the dictionary and the intermediate text. The intermediate text has not changed since it was created by Phases 10D and 10E. The dictionary has been modified by Phase 12. Figure 7 illustrates the data flow within the phase.

Chart 08 illustrates the overall logic of Phase 14 and the relationship among its routines. Table 11, the routine directory, lists the routines used in the phase and their functions.

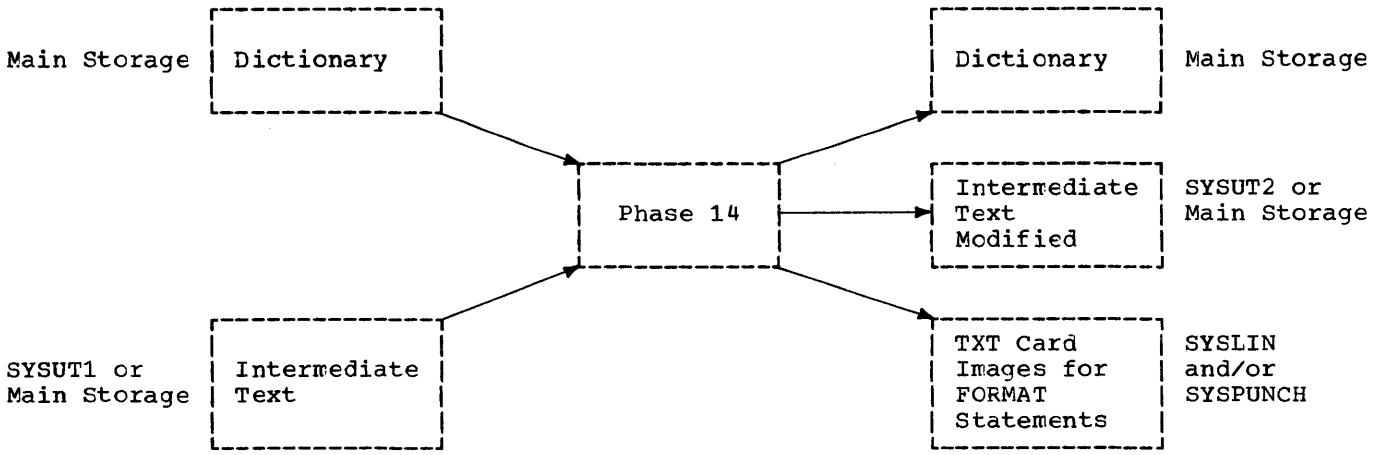


Figure 7. Phase 14 Data Flow

FORMAT STATEMENT PROCESSING

A FORMAT statement is composed of one or more format specifications that define an I/O format. For a discussion of the physical structure of a FORMAT statement refer to the publication IBM System/360 Operating System: FORTRAN IV (E) Language.

Each FORMAT statement is examined beginning with the first FORMAT code. For each FORMAT code obtained, a specific processing routine is called (refer to Table 10). The processing of each routine consists of entering the required information for the FORMAT code into TXT card images. These images are composed of 1-byte units containing 2 hexadecimal digits. Each byte contains one of the following:

- An adjective code, which indicates to IHCFCOME the format conversion (H,I,F,P,X, etc.), a group or field count, or the end of a FORMAT statement.
- A number that represents the actual field count, field length, group count, or decimal length.

One of the following is entered into a TXT card image:

- Adjective Code and Number. Entered for FORMAT specifications P,I,T,A, and X, and for entries made to indicate a field or group count.
- Adjective Code. Entered for a slash, the right parenthesis that ends a group, or the right parenthesis that ends a FORMAT statement.
- Adjective Code, Field Length, and Decimal Length. Entered for FORMAT specifications D, E, and F.
- Adjective Code, Field Length, and Literal. Entered for FORMAT specifications H and apostrophe.

As the specific information is entered into TXT card images, addresses are assigned by incrementing the location counter (according to the amount of storage required to contain the contents of a TXT card image).

During the processing of a FORMAT statement, various accumulators are used to determine the record length. That length is compared to the user-specified length (indicated by the LINELNG option). If the record length is greater than the specified length, a warning indicator is placed in intermediate text. If the user has not specified a record length, the standard length is used.

READ/WRITE STATEMENT PROCESSING

The READ/WRITE statement processing involves four operations. The first is a check for the validity of the symbol used for the data set reference number. An indicator for the end of the READ/WRITE statement is made by entering an end-of-statement indicator in the intermediate text before any entries for the I/O list. This allows Phase 20 to handle the I/O list as a separate statement in intermediate text.

The second operation is the replacement of dictionary pointers in intermediate text (for the symbols in the I/O list) with addresses assigned by Phase 12. This includes a check for the validity of the symbols in the I/O list. When an invalid symbol (a symbol other than a variable or array name) is encountered, an error condition is noted in the intermediate text and the remainder of the I/O list is deleted.

The third operation is to check for and process implied DOs, which are recognized by a left parenthesis within a READ/WRITE statement. For each encounter, an implied DO adjective code is inserted in the intermediate text for the READ/WRITE statement. When the end of an implied DO is recognized (right parenthesis), an end DO adjective code is inserted in the intermediate text.

The fourth operation is to rearrange the READ/WRITE statement entries so that later phases can process the statement correctly. The implied DO variable and parameters are placed ahead of any subscripted variables (whose intermediate text is also rearranged).

REPLACING DICTIONARY POINTERS

In the intermediate text entries for FORTRAN statements, other than the END and FORMAT statements, dictionary pointers are replaced by:

- The address assigned and placed in the dictionary chain field by Phase 12 if the pointer refers to an entry for a variable, constant, array, or external function. (The assigned addresses are obtained from the chain address fields of the affected entries in the dictionary.)
- A data set reference number if the pointer refers to a data set reference number.
- A statement function number if the pointer refers to a statement function.

MISCELLANEOUS STATEMENT PROCESSING

Statement function (SF) definition statements are assigned a unique SF number by Phase 14. This number is used to reference the SF within an associated branch list table in the compiled source module (refer to Phase 25). This unique number is assigned, in sequence beginning with 01, to each SF in the program and is moved to the dictionary entry for the name of that SF. This number also replaces the pointer field of the intermediate text entry for the SF.

The text for RETURN, DO, GO TO, IF, PAUSE, and STOP statements is examined to determine if the statement in question ends a DO loop. If it does, an error condition is noted in the intermediate text. In addition to this error check, if the adjective code for a RETURN statement appears within a main program, that adjective code is changed to the adjective code that represents a STOP statement.

A statement number entry in the intermediate text, other than a FORMAT statement number, is moved unchanged from the input buffer to the output buffer. A FORMAT statement number is treated as follows:

- If the number is not referenced, a warning condition is noted in the intermediate text.
- If the number is associated with a FORMAT statement that ends a DO loop, an error condition is noted in the intermediate text.
- If neither a warning nor error condition is noted for the number, the contents of the location counter are entered in the chain address field of the associated overflow table entry.

BACKSPACE, REWIND, and END FILE statements are examined to verify that the data set reference number is a valid symbol.

Intermediate text for computed GO TO statements is rearranged, putting the variable and the number of statement numbers before the statement numbers themselves.

PHASE 15 (IEJFPAA0)

Phase 15, the fifth processing phase of the compiler, is entered either after the completion of Interlude 14 for SPACE compilations, or after the completion of Phase 14 for PRFRM compilations. The functions of the phase are:

- Reordering intermediate text.
- Modifying intermediate text.
- Assigning registers.
- Creating argument lists.
- Checking for statement errors.

All of the above functions are performed for the processing of statements that can contain arithmetic expressions; only the error checking function is performed for the remaining statements.

Phase 15 reorders the sequence of intermediate text words within statements that can contain arithmetic expressions (arithmetic, arithmetic IF, CALL, and statement functions) so that the resulting object code generated by Phase 25 will cause evaluation of arithmetic expressions according to a hierarchy of operators. As intermediate text words are being reordered, they are modified, depending on the operators and operands, to a form closely resembling an instruction format. When the intermediate text words are modified, registers are assigned, when necessary, to the operands of all arithmetic operators. Argument lists for subprogram and statement function references are created, and in-line function references are processed by generating the appropriate instruction format intermediate text or intermediate text word for an in-line function call. During the input text processing, errors pertaining to DO loops, arithmetic IF statements, statement numbers, function arguments, and operand usage and form are recognized, and the appropriate error messages are given.

Upon completion of Phase 15 processing, control is passed either to Interlude 15 (IEJFPGA0) for SPACE compilations, or to Phase 20 for PRFRM compilations.

Figure 8 illustrates the data flow within Phase 15.

Chart 09 illustrates the overall logic of Phase 15 and the relationship among its routines. Table 14, the routine directory, lists the routines of the phase and their functions.

REORDERING INTERMEDIATE TEXT

Phase 15 reorders the sequence of intermediate text words within arithmetic expressions so that the resulting code generated by Phase 25 will cause evaluation of arithmetic expressions according to a hierarchy of operators. The desired order is defined by a hierarchy of the specific operations as represented by adjective codes and is determined by a comparison of forcing values (a forcing value indicates an operator's priority in the hierarchy of operators). (Refer to Appendix B, Figure 18, for a list of the various operators and their corresponding forcing values.) Depending on the operator in an intermediate text word and its relative position in the hierarchy of operators, that intermediate text word is either:

- Processed (this consists of modifying the intermediate text word by replacing the adjective code field and the mode/type code field, when necessary, with a machine operation code and a register number, respectively), or

- Stored in an operations table or subscript table (refer to Appendix B, Figures 19 and 20).

The operations and subscript tables function as pushdown tables in which the top entry in the table is the most recently entered item. (This process is known as LIFO: last in, first out.)

The actual reordering of intermediate text words is controlled by a routine (FOSCAN) that scans the input intermediate text words. This routine compares the forcing values of the various adjective codes under consideration to determine their disposition. Each adjective code has a left and a right forcing value. The right forcing value applies to the adjective code within the current input intermediate text word. The left forcing value applies to the adjective code within the

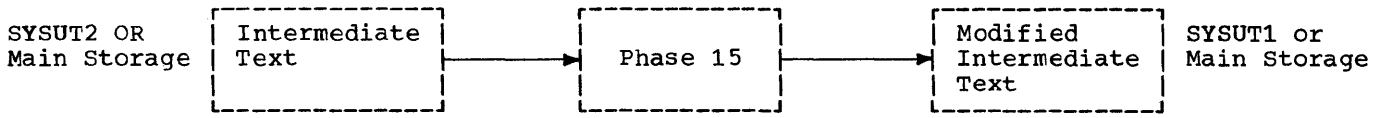


Figure 8. Phase 15 Data Flow

top entry in the operations table. The adjective code of the first intermediate text word of an arithmetic statement has the highest left forcing value of any adjective code except for the end-of-statement indicator.

The first intermediate text word of any arithmetic statement is first written on the output data set and then entered in the operations table. The next word of the input intermediate text for this statement is then obtained and examined. If it is subscript intermediate text, it is entered in the subscript table. The following word is then obtained and examined. When the word (in the operations table) containing the subscripted variable is processed, the related subscript intermediate text is obtained from the subscript table. The related subscript intermediate text is always the top entry in the subscript table.

If the word obtained from the input intermediate text is not a subscript intermediate text word, the right forcing value of that word is compared to the left forcing value of the top entry in the operations table. If the right forcing value is greater than or equal to the left forcing value, the top entry of the operations table is forced out, processed, and written on the output data set. If the right forcing value is less than the left forcing value, the current word of the input intermediate text is entered into the operations table. The next input intermediate text word is then obtained. This comparison process continues until the first entry (for the statement under consideration) made in the operations table is forced out (by the end mark) and processed. In this way, the input data set is reordered when it leaves Phase 15 as the output data set.

If an attempt is made to enter information in the operations or subscript table when they are full, an error condition is recognized. An error intermediate text word, which indicates that the statement is too long and should be subdivided, is generated and placed at the end of the intermediate text words for the statement containing the error.

MODIFYING INTERMEDIATE TEXT

As intermediate text words are being reordered, they are modified, depending on the operators and operands, to a form closely resembling an instruction format. The contents of the adjective code field for arithmetic operators (unary minus (ū), +, -, *, and /) are replaced by the appropriate machine operation code. The contents of the mode field are replaced by a register number when the operator and operands require a register assignment.

ASSIGNING REGISTERS

Registers are assigned by Phase 15 according to the adjective code encountered and the mode of the operands. There are eight registers (general registers 0, 1, 2, and 3; floating-point registers 0, 2, 4, and 6) that may be assigned by Phase 15. When a register is required for a particular operation and one is not available, the contents of the required register are transferred to a work area. That register acquires "available" status and is then used for the operation.

Register assignments are made by Phase 15 according to the following rules:

- The instruction generated for the add operator and the floating-point multiply operator requires that one of its operands be in a register. The instruction generated for the multiply operator for integer quantities requires that the multiplicand (left operand) be in an odd register. The even register that precedes the multiplicand must be made available, unless it already contains the multiplier.

- The instruction generated for the subtract operator and the divide operator for real quantities requires that its left operand be in a register.

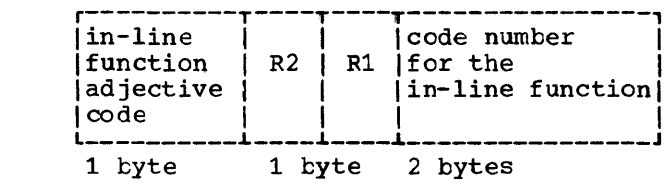
- For integer division, the dividend must be in an even-odd register pair.

- A work register is assigned to each subscript expression to aid in the computation of subscript expressions by Phase 20.

- Exponentiation requires library subprograms; therefore, a specific register is required to contain the result of the subprogram execution.

- Registers are assigned to single and double in-line functions, as follows:

  There are eight single-argument, in-line functions: IFIX, FLOAT, DFLOAT, SNGL, DBLE, ABS, IABS, and DABS. Instructions are generated to perform the functions of the SNGL and DBLE in-line functions. For the remaining single-argument, in-line functions, a word in the following format is generated:



Depending upon the specific in-line function, up to three registers are assigned by Phase 15. For ABS, IABS, and DABS, only an argument register is required. This register is indicated in the R1 field; the R2 field is made zero. For IFIX, FLOAT, and DFLOAT, three registers are required: an argument register, a result register, and a work register. The argument register is indicated in the R1 field, the result register in R2. The work register is the register preceding R1.

For in-line functions with two arguments, an in-line call word is generated with the same format as for single-argument, in-line functions. Phase 15 assigns a register to each argument in a double-argument, in-line function. The first argument register is indicated in the R1 field; the second argument register is indicated in the R2 field. R1 is used as the result register.

CREATING ARGUMENT LISTS

To assist Phase 25 in the generation of the object module instructions, a list of arguments is created when an adjective code is encountered that represents a call to a subprogram or to a statement function. The argument list is preceded by an intermediate text word that defines the specific function call. The first word of the argument list contains the number of arguments in the list, and is followed by an intermediate text word for each argument. The total number of arguments in all lists created by Phase 15 is kept in the communication area to be used by Phase 20 processing.

CHECKING FOR STATEMENT ERRORS

As each statement is processed, Phase 15 checks for specific error conditions. General format errors as well as specific errors connected with DO statements, arithmetic IF statements, statement numbers, and argument lists are noted. Following are the error checks performed by Phase 15:

- DO loops are examined to determine if the DO variable is redefined, or if a DO loop is nested to a depth greater than 25.
- Arithmetic IF statements are examined to determine if the arithmetic expressions contain valid symbols. They are also examined to determine if more or fewer than three statement numbers have been specified.
- Statement numbers are examined to ensure that they are uniquely defined and do not indicate transfers to nonexecutable statements.
- If a FUNCTION subprogram is being compiled, a check is made to determine whether the subprogram name is defined.
- The members of an argument list are examined to determine whether they are valid. If a particular list has a required length, that list is examined to determine if it is of the required length.

If any of the designated error conditions are encountered, an intermediate text word, which contains an adjective code indicating an error and a number representing the specific error, is generated and placed at the end of the intermediate text words for the statement in which the error was detected.

PHASE 20 (IEJFRAA0)

Phase 20, the sixth processing phase of the compiler, is entered either after the completion of Interlude 15 for SPACE compilations, or after the completion of Phase 15 for PRFRM compilations. The major functions of the phase are:

- Processing of statements that require subscript optimization.
- Processing of statements that affect, but do not require, subscript optimization.
- Creating the argument list table.

Phase 20 increases the efficiency of the object module by decreasing the amount of computation associated with subscript expressions. A subscript expression can recur frequently in a FORTRAN program. Recomputation at each occurrence is time-consuming and results in an inefficient object module. Therefore, Phase 20

performs the initial computation of any given subscript expression and assigns a register which, at object time, contains the result of this computation. Phase 20 then modifies (that is, optimizes) the intermediate text for subsequent occurrences of this subscript expression. This intermediate text optimization consists essentially of replacing the computation of the subscript expression, at each recurrence, with a reference to its initial value (that is, to the register that contains the result of the initial computation). The subscript intermediate text for each subsequent occurrence of the subscript expression can be optimized in this manner as long as the values of the integer variables in the expression remain unchanged.

In addition, the following functions are performed by Phase 20:

1. Generation of ESD card images for:
   a. Implied external references to required library exponentiation subprograms.
   b. Implied external references to IHCFCOME (that is, IBCOM#).
   c. Implied external references to IHCCGOTO (that is, CGOTO#). (IHCCGOTO is an implicitly called library subprogram that aids in the execution of computed GO TO statements by supplying the object-time branch addresses.)
2. Generation of TXT and RLD card images for literals generated by Phase 20 and argument list table entries.
3. Generation of calling sequences to IHCIBERR (that is, IBERR#) when source statement errors are encountered. (Refer to Appendix G for a description of the IHCIBERR object-time library subprogram.)
4. Printing of a storage map for all literals generated by Phase 20, and for all implied external references made by the source module being compiled, if the MAP option is specified.
5. Allocation of storage for the branch list table for SF expansions and DO statements.

Upon completion of Phase 20 processing, control is passed either to Phase 30 (if the NOLOAD option was specified and source module errors were detected), or to Phase 25.

Figure 9 illustrates the data flow within Phase 20.

Chart 10 illustrates the overall logic and the relationship among the routines of Phase 20. Table 17, the routine directory, lists the routines used in the phase and their functions.

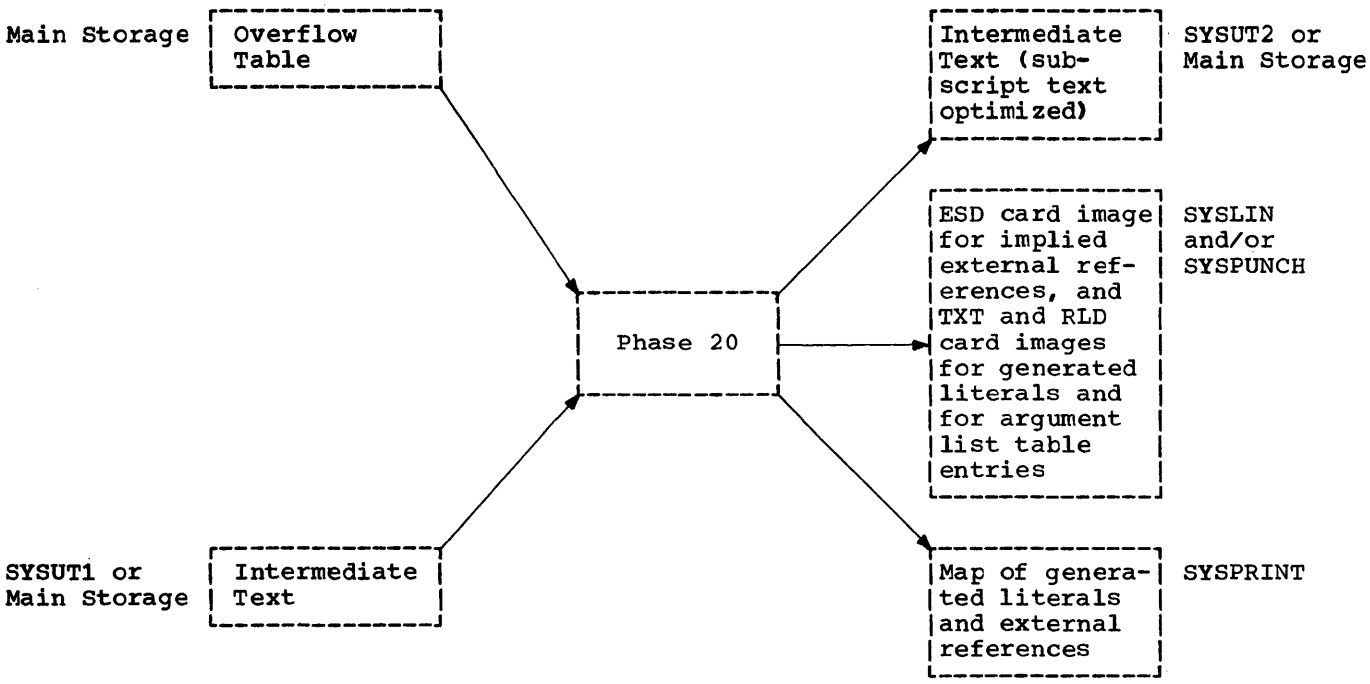


Figure 9. Phase 20 Data Flow

PROCESSING OF STATEMENTS THAT REQUIRE SUBSCRIPT OPTIMIZATION

Phase 20 scans the input text for statements that may require subscript optimization. Subscript expressions may occur in the following statements:

- Arithmetic.
- CALL.
- Arithmetic IF.
- Input/output lists (input/output lists are treated as statements by Phase 20).

When Phase 20 encounters one of these statements containing a subscripted variable, the subscript optimization process begins.

An index mapping table (refer to Appendix B, Figure 21), containing all information pertinent to a subscript expression, is used to aid subscript processing. When the index mapping table indicates the first occurrence of the current subscript expression, a register is assigned and a corresponding entry is made in the index mapping table. When a register is not available, the register that is currently assigned to the subscript expression of least dimension is reassigned to the current subscript expression.

If the current subscript expression has been encountered previously, the intermediate text for its computation can be replaced effectively by a reference to the register assigned at the first encounter. However, redefinition of any integer variable in the expression invalidates the previous computation and prohibits the assignment of the same register to the current subscript expression. In this case, recomputation is necessary and another register must be assigned for the subscript expression.

During the subscript optimization process, Phase 20 may be required to generate literals connected with the array displacement associated with any given subscript expression. (Refer to Appendix E for a discussion of the calculation of an array displacement. This explanation includes a description of the offset and CDL (constant, dimension, and length) portions of an array displacement.) Literals are generated by Phase 20 under the following conditions:

- When the optimization routine encounters a value outside the addressable range of 0 through 4095 bytes as a result of adding the offset (calculated in Phase 10E) to the displacement of the array variable (calculated in Phase 15), an offset literal is generated. The generation of an offset literal allows Phase 25 to produce instructions involving these subscripted variables without having to assign a new base register.
- Phase 20 generates a literal for each component of the CDL portion of the array displacement associated with a subscript expression except for the first component if it is a power of 2. In this case, that power, instead of the address for the literal C1*L, is placed in the subscript text.

The preceding discussion of subscript optimization applies to subscript expressions that are neither constant nor associated with a dummy subscripted variable. These two conditions are discussed in the following paragraphs.

Phase 20 does not assign a register to a constant subscript expression which, when added to the offset portion of the array displacement, lies within the addressable range of 0 through 4095 bytes. However, if this computation lies outside the above range, a register is assigned for this constant and an entry is made in the index mapping table.

In addition to normal optimization, a base register is assigned to any dummy variable so that the variable may be addressed during execution of the object module. This assignment is entered in the index mapping table.

PROCESSING OF STATEMENTS THAT AFFECT, BUT DO NOT REQUIRE, SUBSCRIPT OPTIMIZATION

In addition to previously mentioned statements that require subscript optimization, various other statements that can affect the subscript optimization process are processed by Phase 20.

DO and READ Statements

The DO and READ statements sometimes cause the redefinition of the integer variable(s) in a subscript expression. Any integer variable that is redefined becomes a bound variable. Any encounter of a bound variable causes Phase 20 to examine the subscript expressions that are assigned registers in the index mapping table. A bound variable in a subscript expression invalidates any previous computation for that expression and causes a new register to be assigned for that expression.

Referenced Statement Numbers

When a statement number is referred to in other statements (for example, a GO TO statement), Phase 20 does not know if the values of previously encountered integer variables can still be used by subscript expressions containing these variables. Because any given variable may now be a bound variable, Phase 20 deletes all register assignments (in the index mapping table) for subscript expressions involving that variable.

Subprogram Argument

Any subprogram argument that is an integer variable causes redefinition of that variable and, therefore, invalidates any previous computations of subscript expressions containing that variable. All register assignments (in the index mapping table) for subscript expressions involving that variable are deleted.

CREATING THE ARGUMENT LIST TABLE

A count of the number of arguments contained in the source module for subprogram and SF (statement function) calls is passed to Phase 20 via the communication area. This number is used by Phase 20 to allocate storage for the argument list table. Phase 20 allocates a word (4 bytes) for each argument, and inserts the relative address of each argument in the argument list table.

If an argument is a subscripted variable, its address is not known at this time. Instructions are generated to load the address of this argument into the argument list table at object-time.

The table is used at object-time to provide the addresses of argument lists to the subprograms and SFs being called. Refer to Appendix F, Figure 45, for the format of the argument list table.

For each subprogram name or SF name encountered, Phase 20 generates the appropriate calling sequence. A register is used to supply the referenced subprogram or SF with the address of its argument list. Phase 20 also generates RLD and TXT card images for each entry in the argument list table.

PHASE 25 (IEJFVAA0)

Phase 25, the seventh processing phase of the compiler, is entered after the completion of Phase 20. The main functions of the phase are:

- Generation of object module instructions.
- Completion of object module tables.

Phase 25 creates the object coding for the FORTRAN source module from the intermediate text entries and the overflow table (refer to Appendix C). TXT card images for instructions are generated and then written on the SYSLIN data set (if the LOAD option is specified) and/or the SYSPUNCH data set (if the DECK option is specified).

Several tables (branch list table for statement numbers, branch list table for SF expansions and DO statements, and base value table) are used by the object module during execution of the instructions generated by Phase 25. These tables are assembled in their final form by Phase 25.

In addition to the above functions, Phase 25 generates: (1) a listing of referenced statement numbers if the MAP option is specified, and (2) an object module listing if the object listing option is specified and if the object listing facility of the compiler has been enabled. The object module listing contains the machine language instructions generated by Phase 25 and their equivalent assembly language instructions. The equivalent assembly language instructions are generated by an object listing module (IEJFVCA0) that Phase 25 loads (via the LOAD macro-instruction) into main storage.

Upon completion of Phase 25 processing, control is passed to Phase 30 (to generate error/warning messages and to process the END statement).

Figure 10 illustrates the data flow within Phase 25.

Chart 11 illustrates the overall logic and the relationship among the routines of Phase 25. Table 19, the routine directory, lists the routines used in the phase and their functions.

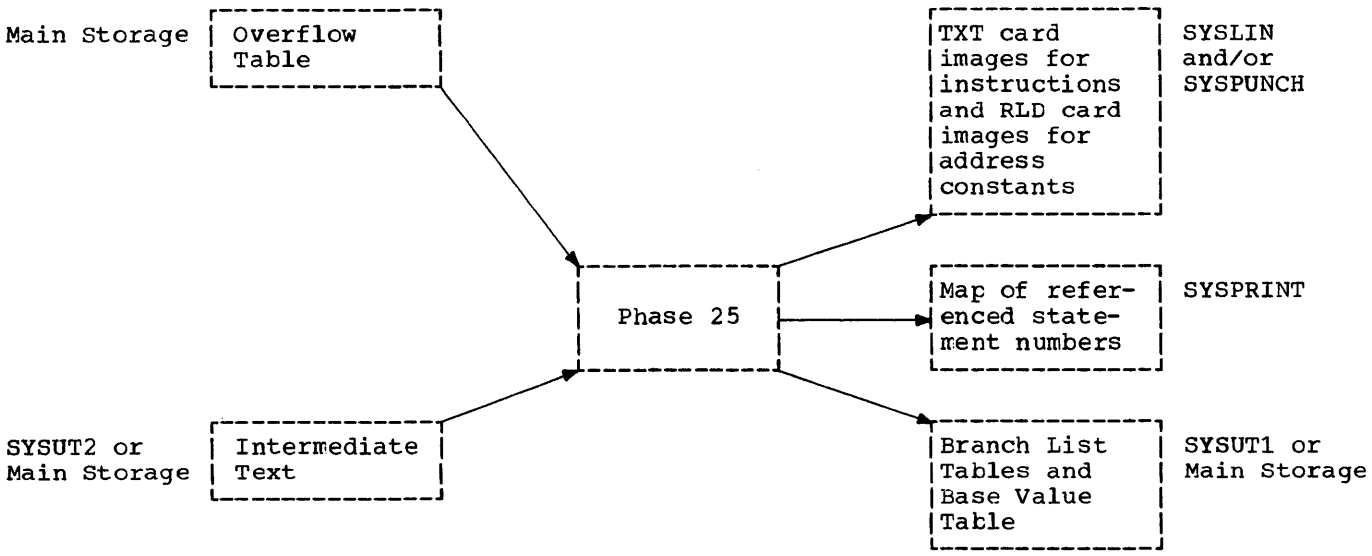


Figure 10. Phase 25 Data Flow

GENERATION OF OBJECT MODULE INSTRUCTIONS

Phase 25 creates the object module instructions for the FORTRAN source module from the intermediate text entries and the overflow table. The resultant object module instructions are in the RR, RX, and RS formats of the System/360 instructions.

The control routine (PRESCN) for Phase 25 obtains each intermediate text entry and examines its adjective code. The adjective code determines which Phase 25 subroutine is to process the current entry or the next series of entries. The processing subroutine generates the required object coding.

Intermediate text entries for operations within arithmetic expressions are almost in a final instruction format as a result of Phase 15 processing. The intermediate text words generated by Phase 15, for arithmetic expressions, contain all the elements required for the RX format instruction: operation code, result register, base register, and displacement. When Phase 25 encounters an adjective code indicating an arithmetic expression, control is passed to the routine (RXGEN) that generates RX format instructions.

Other intermediate text entries still resemble the output generated by Phase 14. An adjective code identifies the type of entry and possibly several entries that follow it. Various Phase 25 subroutines analyze these entries and generate the appropriate instructions.

If a subprogram is being compiled, Phase 25 generates an epilog table when the FUNCTION or SUBROUTINE adjective code is encountered. The epilog table provides Phase 25 (when it encounters the RETURN statement) with the information necessary for the generation of instructions that return the new values of variables, used as parameters, to the calling program. This information consists of the following:

- Length and address of the variable in the subprogram.
- The relative position of the variable in the parameter list of the calling program.

Refer to Appendix B, Figure 22, for the format of the epilog table.

COMPLETION OF OBJECT MODULE TABLES

Several tables are used by the object module during the execution of the instructions generated by Phase 25. These tables, assembled in their final form by Phase 25, are:

- The branch list table for referenced statement numbers.
- The branch list table for SF expansions and DO statements.
- The base value table.

Branch List Table for Statement Numbers

Phase 12 allocated storage for a branch list table (refer to Appendix F, Figure 43) for referenced statement numbers. Each statement number referenced by a GO TO, computed GO TO, IF, or DO statement was assigned a number relative to the start of the branch table. This relative number was placed in the chain field of the statement number entry in the overflow table (refer to Appendix C).

When an intermediate text entry for a statement number definition is recognized by Phase 25, the corresponding overflow table entry is obtained, and the relative number, assigned by Phase 12, is used to determine the position of the statement number in the branch table. The value of the location counter is placed in this position and is the actual relative address of that statement.

Two instructions are generated for the portion of a FORTRAN statement that references a statement number. The first instruction loads the address portion of the proper entry in the branch table into a general register; the second instruction branches to the address placed in that general register.

Branch List Table for SF Expansions and DO Statements

A second branch list table is completed by Phase 25 for statement function (SF) expansions and DO statements. Phase 14 assigned a unique number to each SF and placed this number in the pointer field portion of the intermediate text entry for each SF. Phase 25 uses this number to assign a location in this second branch list table when it encounters an SF adjective code. The address of the first instruction in the SF expansion in question is placed in this location. Any statement referencing this SF uses the number of the SF to obtain this location in the branch list table, and branches to the address in the location (that is, to the beginning of the SF expansion).

Phase 25 also assigns each DO statement a location in this branch list table. The address of the second instruction of the DO loop in question is entered in the proper location. The object module instruction that controls the iteration of the DO loop obtains this location in the branch list, and branches to the address in the location (that is, to the beginning of the DO loop). Refer to Appendix F, Figure 44, for the format of the branch list table for SF expansions and DO statements.

Base Value Table

The base value table (refer to Appendix F, Figure 46) is continually generated by the various phases of the compiler as base registers are required for the object coding. An object module can only use general registers 4, 5, 6, and 7 as base registers. (When the object module is entered at object-time, these registers are initialized from entries in the base value table.) If the base register requirements for the object module extend beyond the four available registers, the base value table is used to take special action.

During compilation (prior to Phase 25), the value for each base register to be used by the object module is inserted in the base value table, regardless of the general register number used as the base register. The first entry in the base value table is the value placed in register 4; the second refers to register 5; etc.

For a source module for which the compiler assigns registers 4 and 5 to reference data in COMMON and assigns registers 6, 7, and 8 to reference data and instructions in the object module, the base value table contains the following values:

Register	4	5	6	7	8
Value	0	4096	0	4096	8192

The value 8192 is initially assigned to general register 8, and that register number is entered in the intermediate text entry requiring the base register. However, when Phase 25 encounters this intermediate text entry with a base register number of 8, an instruction is generated to load the value 8192 into register 7, and general register 7 is used as the base register in this instruction.

In general, when a base register other than 4, 5, 6, or 7 is encountered by Phase 25, the base value table is used to obtain the value of that base register, and an instruction is generated to load that value into register 7. Register 7 is used as the base register in the instruction at object time.

PHASE 30 (IEJFXAA0)

Phase 30 is the eighth and last processing phase of the compiler. The phase may be entered either after the completion of Phase 20 processing if the NOLOAD option was specified and errors were detected in the source module or after the completion of Phase 25 processing. The functions of the phase are:

- Producing error and warning messages.
- Processing the END statement.

When Phase 30 is entered from Phase 20, only the first function (producing error and warning messages) is performed. However, when Phase 30 is entered from Phase 25, both functions are carried out.

Upon the completion of Phase 30 processing, control is passed to Phase 1.

Figure 11 illustrates the data flow within Phase 30.

Chart 12 illustrates the overall logic and relationship among the routines of Phase 30. Table 20, the routine directory, lists the routines used in the phase and their functions.

PRODUCING ERROR AND WARNING MESSAGES

Phase 30 checks the adjective code of each intermediate text word for an error or warning condition. If one is encountered, Phase 30 obtains the error or warning number (set up by the phase that detected the error or warning condition) from the mode/type field of that intermediate text word. This number is used as an indexing value to obtain the length and address of the actual message corresponding to the specific error or warning detected.

The length of the message is obtained from the message length table. The address of the message is obtained from the message address table. The actual message is obtained from the message text table. (Refer to Appendix B for a description of the use and format of the message tables.)

When the message length and the message address are obtained, Phase 30 then prints the corresponding message on the SYSPRINT data set. (For a description of the messages capable of being generated by the compiler refer to the publication IBM System/360 Operating System: FORTRAN IV (E) Programmer's Guide.)

PROCESSING THE END STATEMENT

When the intermediate text entry for the END statement is recognized by Phase 25, control is passed to Phase 30. Phase 30 first produces any error or warning messages detected by earlier phases of the compiler. Phase 30 then writes both branch list tables and the base value table onto the output data set(s). Because all three of these tables must be relocatable, all entries in the tables are entered in RLD card images, as well as in TXT card images. Phase 30 also creates the END card image for the object module.

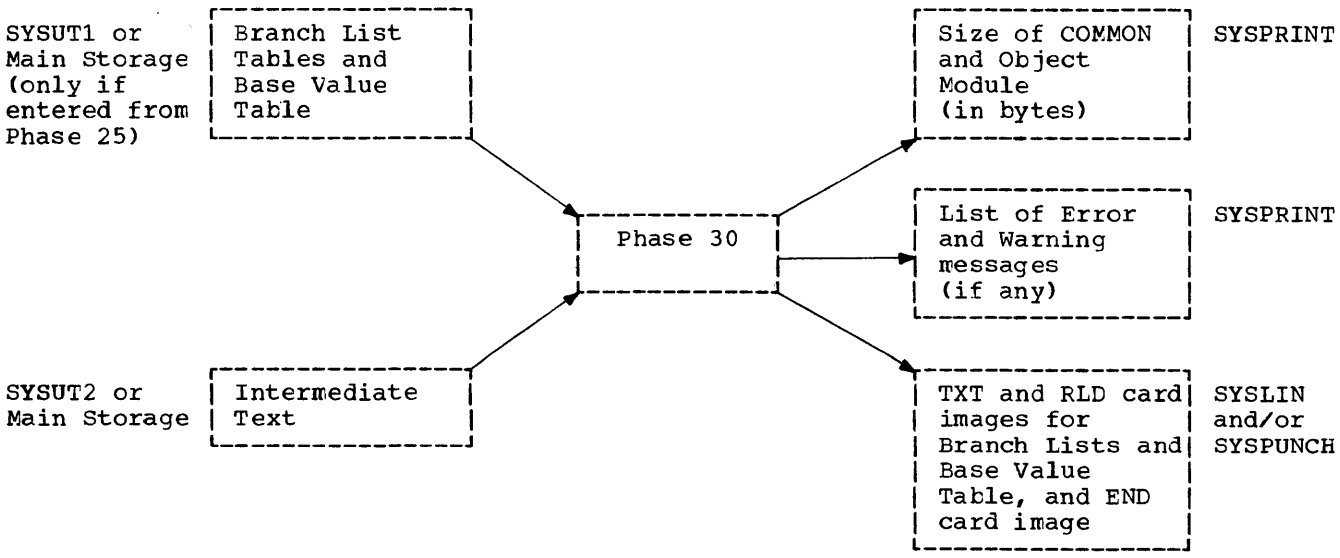


Figure 11. Phase 30 Data Flow

The following charts describe the overall logic of the major components of the FORTRAN IV (E) compiler. Routine directories are included for those components that contain numerous routines and subroutines.

Flowchart Conventions

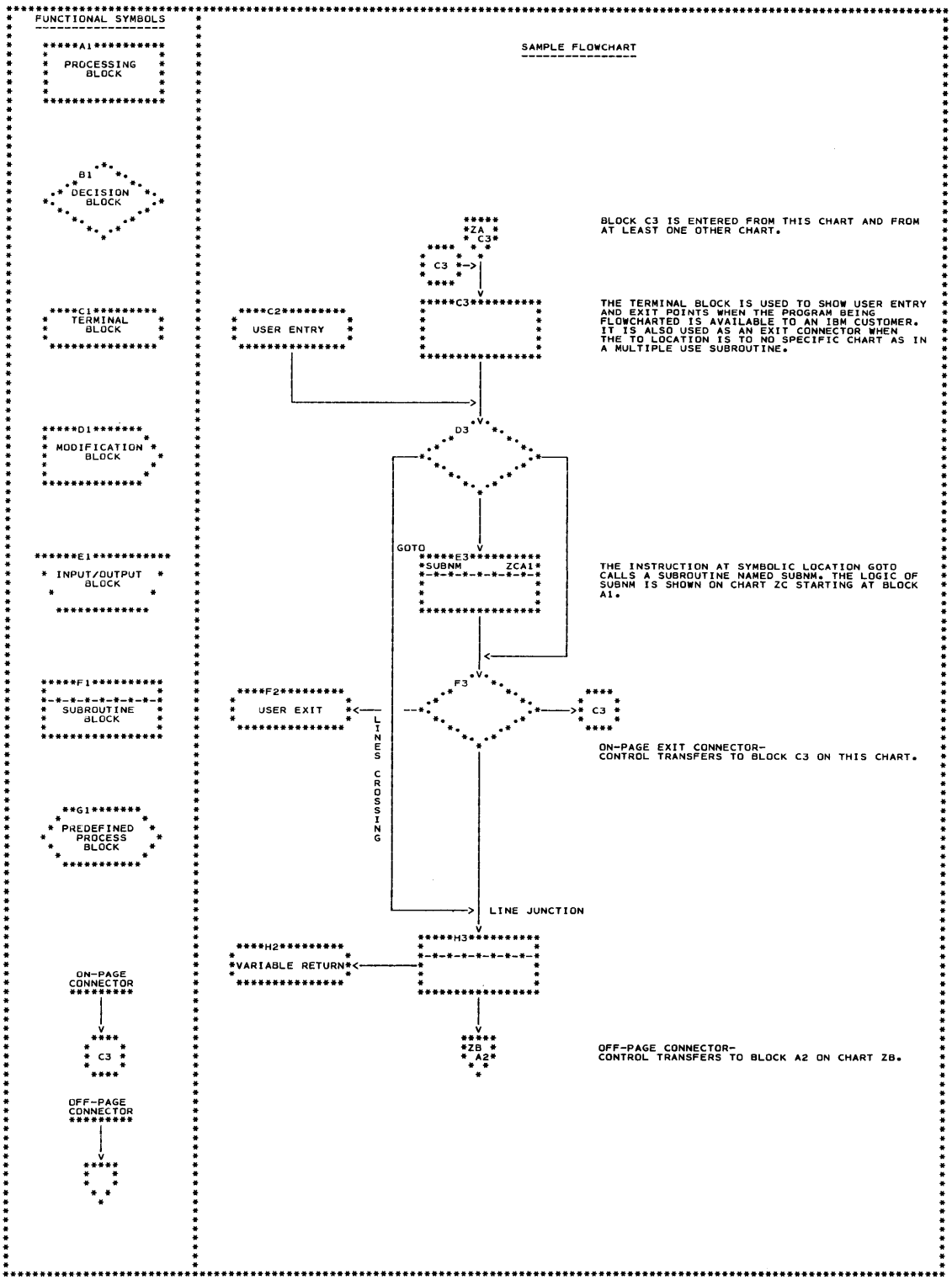


Chart 01. Phase 1 (IEJFAAA0/IEJFAAB0) Overall Logic Diagram

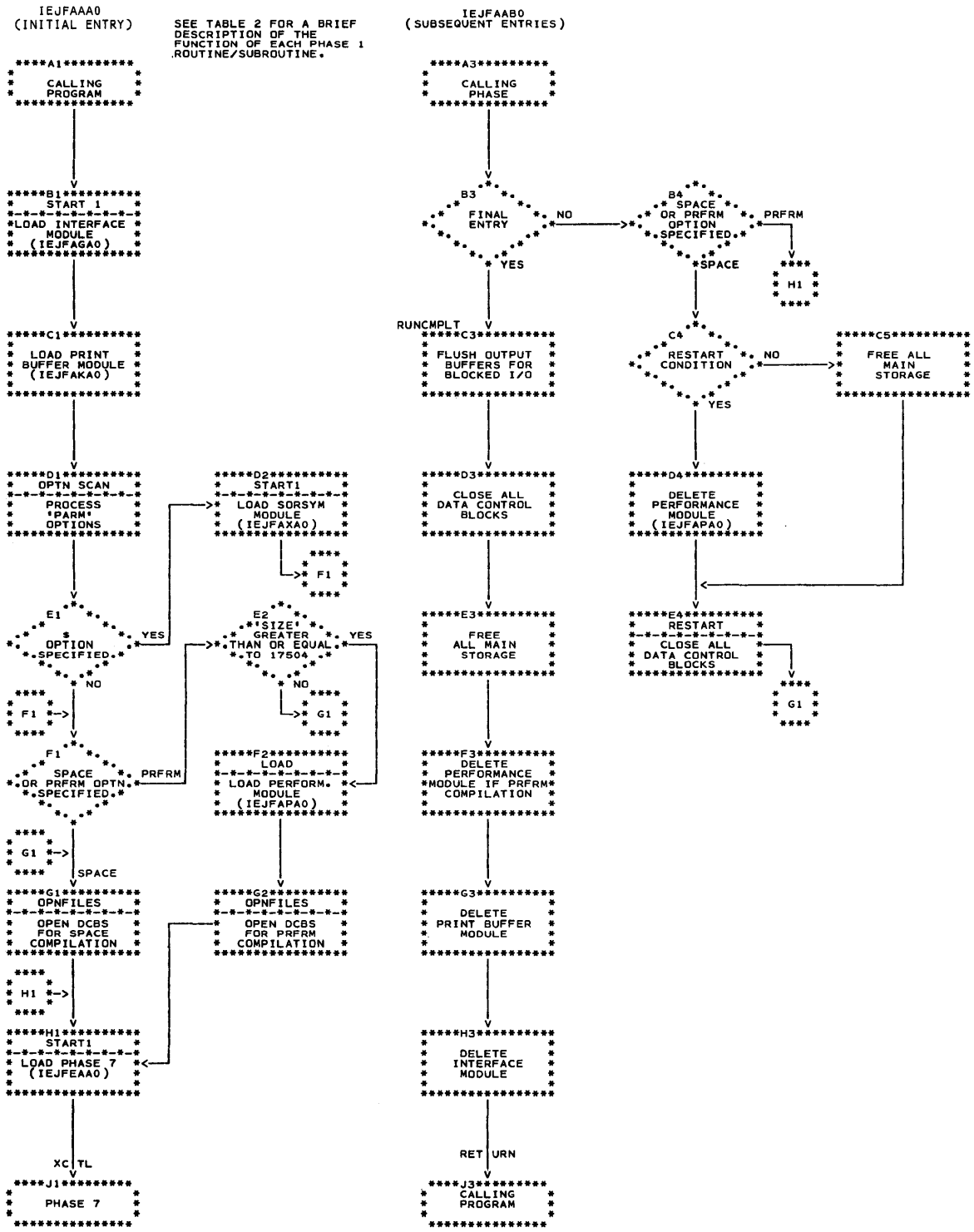


Table 2. Phase 1 Main Routine/Subroutine Directory

Routine/Subroutine	Function
LOAD	Loads the performance module into main storage if the PRFRM option is specified.
OPNFILES	Opens data control blocks for compiler data sets as indicated by switches (in the communication area) for options.
OPTNSCAN	Scans the compiler options and sets appropriate switches in the communication area.
RESTART	Closes all data control blocks for compiler data sets.
RUNCMPLT	Closes all data control blocks for compiler data sets, frees all main storage allocated to the compiler, and returns control to the calling program.
START1	Performs housekeeping and loads the interface module, print buffer module, and Phase 7.

Chart 02. Interface Module (IEJFAGA0) Routines

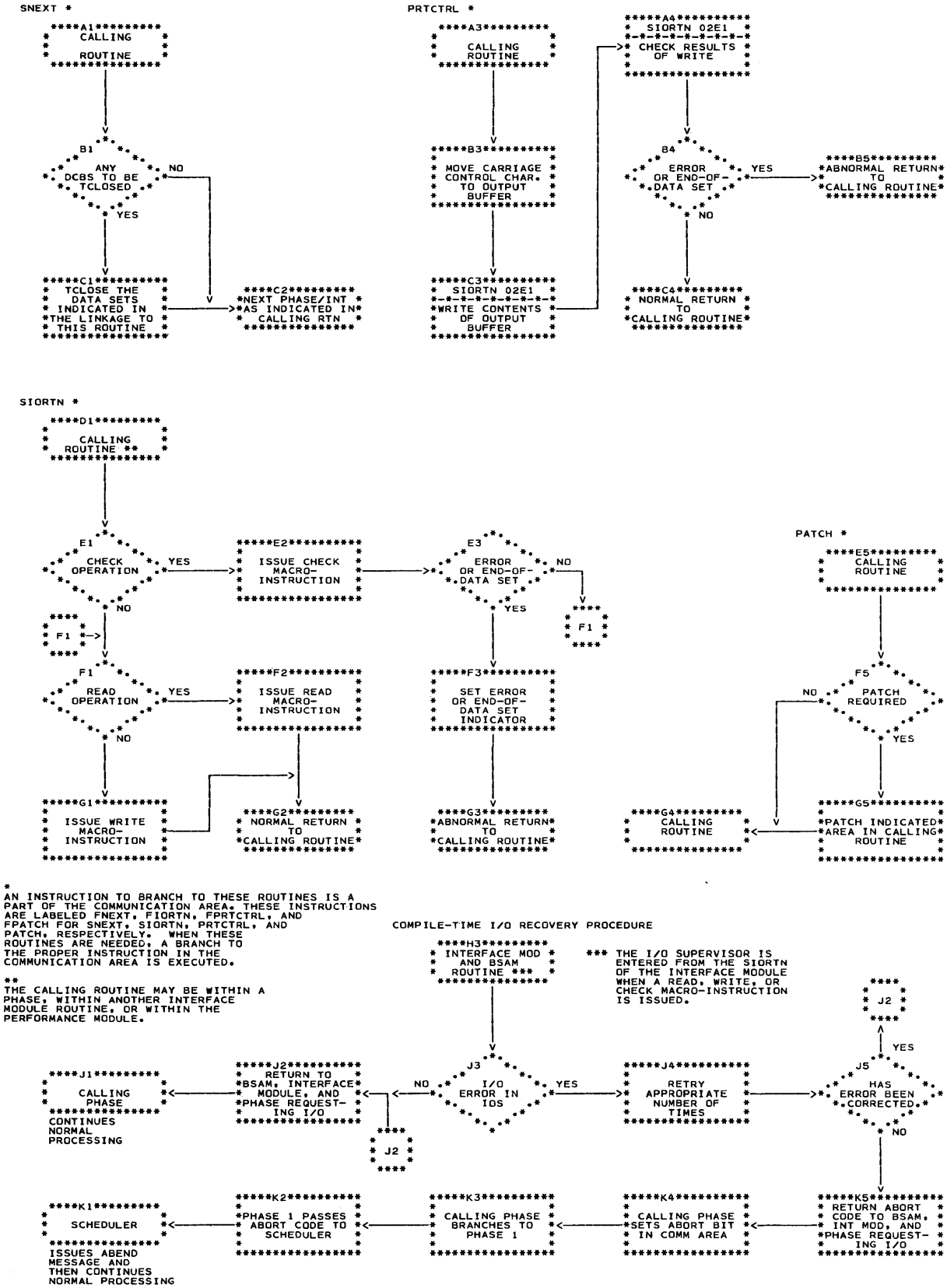


Chart 03. Performance Module (IEJFAPA0) Routines

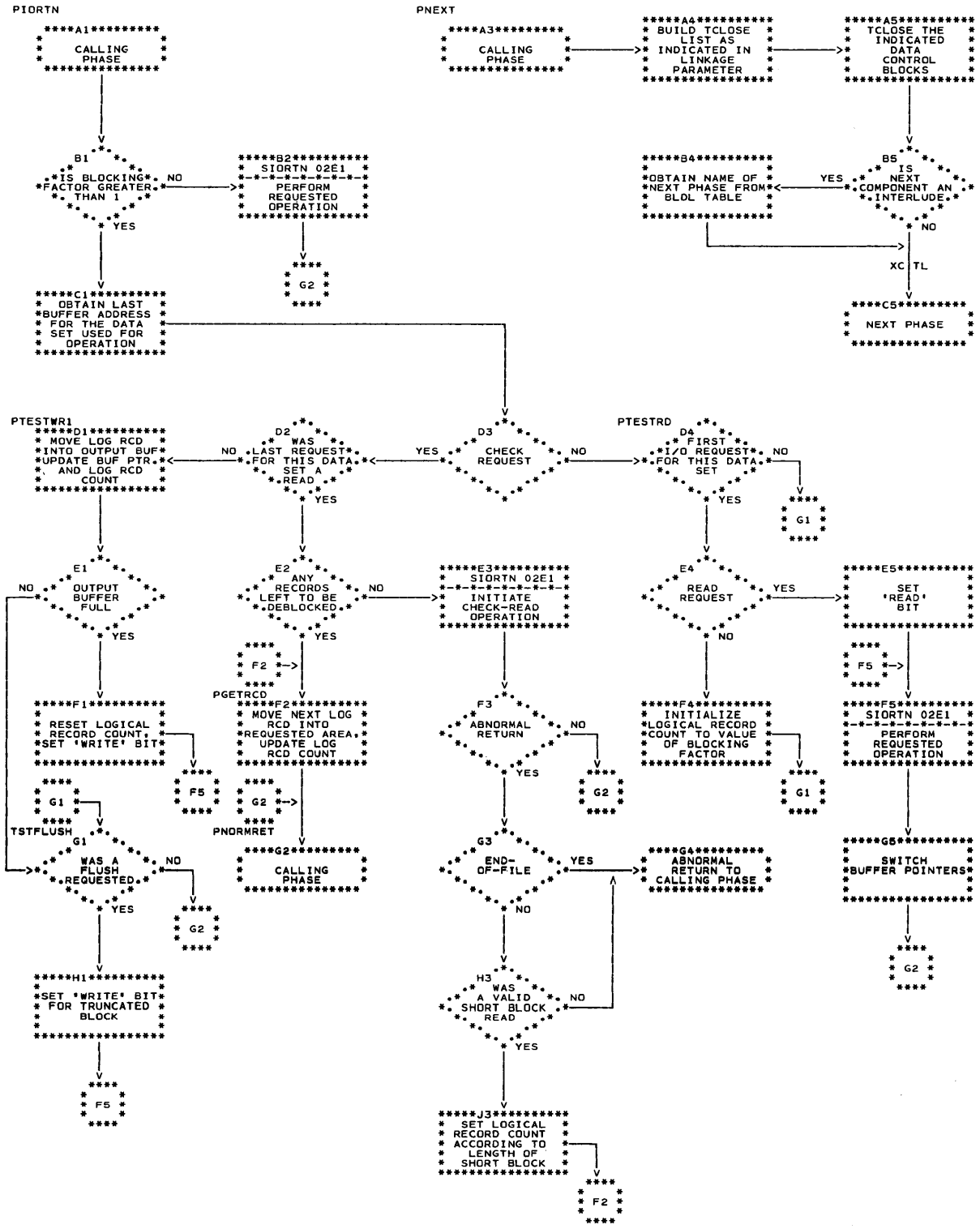


Chart 04. Phase 7 (IEJFEAA0) Overall Logic Diagram

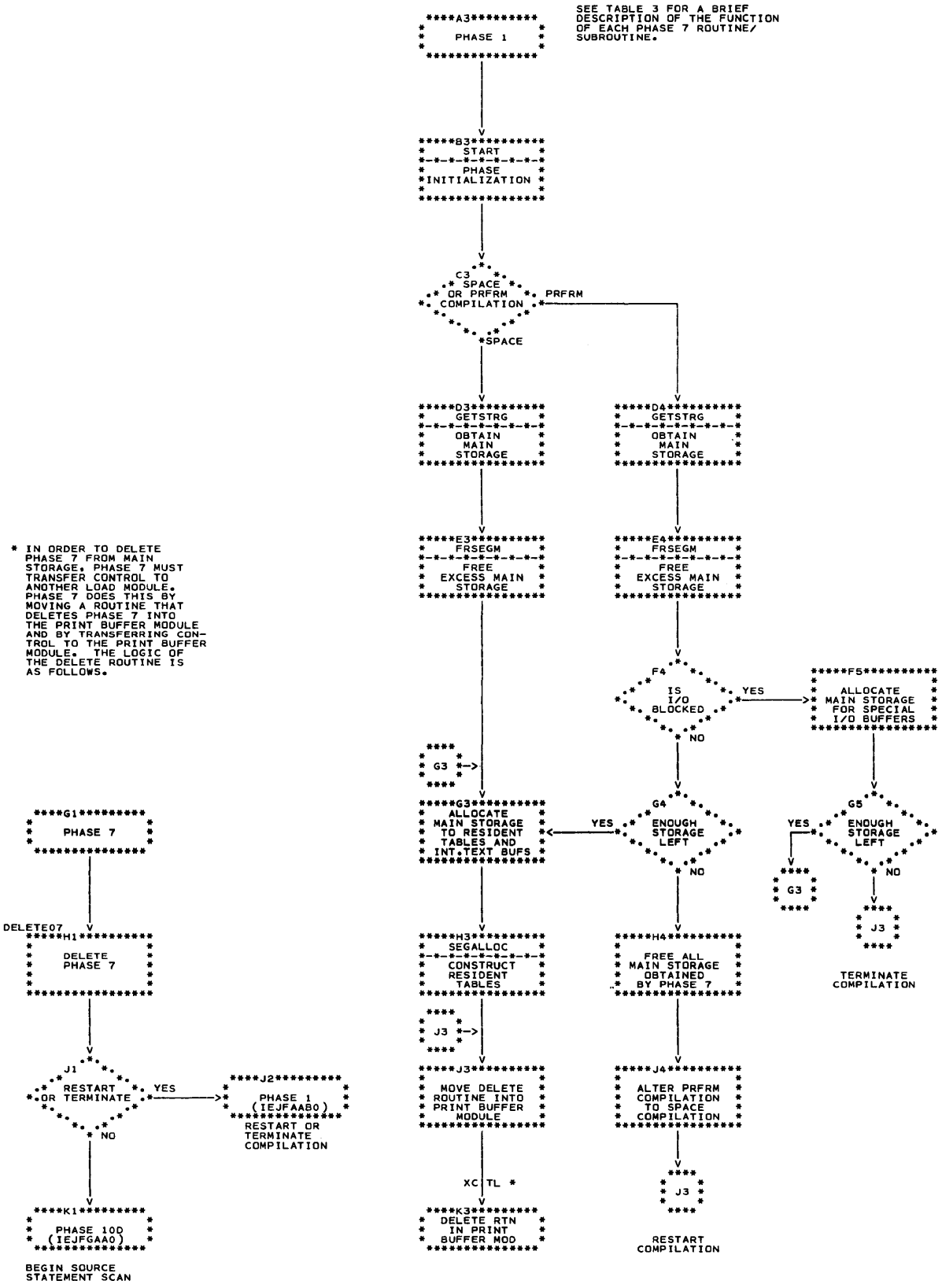
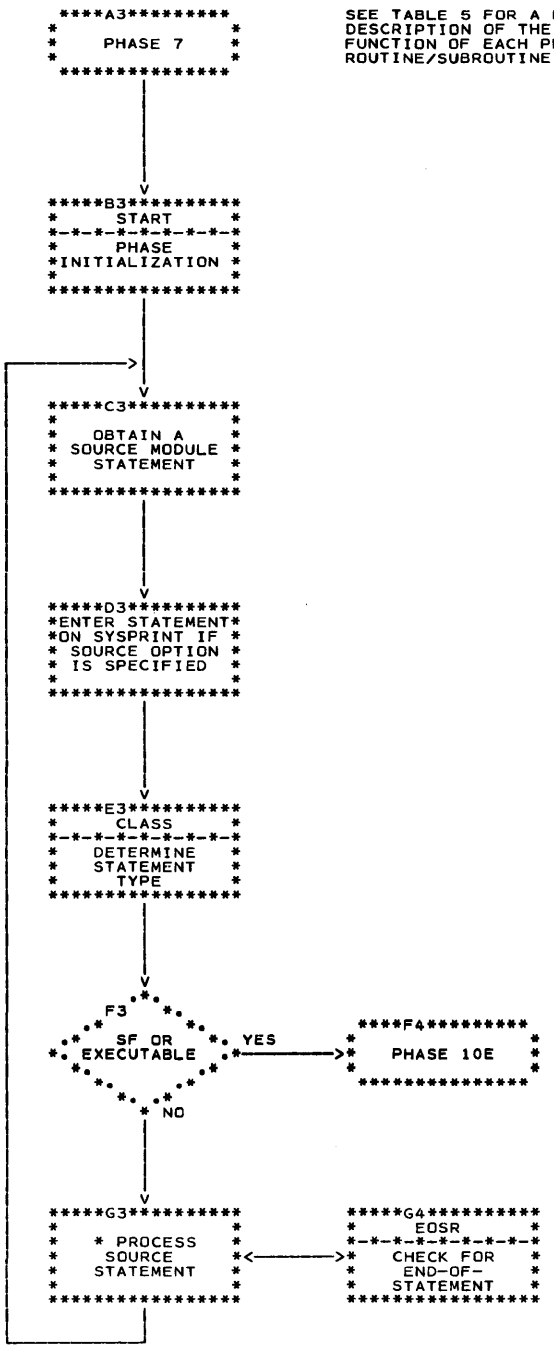


Table 3. Phase 7 Main Routine/Subroutine Directory

Routine/Subroutine	Function
DELETE07	Deletes Phase 7.
FRSEGM	Frees transient work area and any unusable main storage.
GETSTRG	Obtains main storage for the compiler.
MESSGOUT	Writes messages on SYSPRINT.
SEGALLOC	Completes the construction of SEGMAL (begun in GETSTRG), and builds necessary dictionary and overflow table structure (independent of the source module being compiled).
START	Performs Phase 7 initialization.
TEMPATCH	Builds patch table by reading and then converting patch records.

Chart 05. Phase 10D (IEJFGAA0) Overall Logic Diagram



* SEE TABLE 4 FOR A LIST OF THE STATEMENTS PROCESSED BY PHASE 10D AND THE MAIN ROUTINES AND SUBROUTINES THAT PROCESS THESE STATEMENTS.

Table 4. Phase 10D Statement Processing

Statement Type	Main Processing Routine	Main Subroutines Used ++
REAL	REAL/INTGER/DOUBLE **	Control is passed to DIM
INTEGER	REAL/INTGER/DOUBLE **	
DOUBLE PRECISION	REAL/INTGER/DOUBLE **	
DIMENSION	DIM **	GETWD, RCOMA, CSORN, DIMSUB, WARN/ERRET
COMMON	COMMON ** *	DIM, PUTBTXT
EQUIVALENCE	EQUIV ** *	GETWD, CSORN, WARN/ERRET, PUTBTXT, RCOMA
EXTERNAL	EXTERN **	GETWD, RCOMA, CSORN
FUNCTION	FUNCT * **	GETWD, CSORN, PUTX
SUBROUTINE	SUBRUT * **	
FORMAT	FORMAT *	GETWD, WARN/ERRET, PUTX

* Text is created when processing this statement.

** Table entries may be prepared when processing this statement.

++ All routines except FORMAT use ERROR as an error exit for errors that cause termination of the statement processing. FORMAT has nc error exit.

Table 5. Phase 10D Main Routine/Subroutine Directory

Routine/Subroutine	Function
CLASS	Determines which routine will process the statement type. May use LOADE and LABLU.
COMMON	Processes COMMON statements.
CSORN	Processes names, constants, data set reference numbers, and DO parameters. May use LITCON and SYMTLU.
DIM	Processes the variables of DIMENSION, COMMON, INTEGER, REAL, and DOUBLE PRECISION statements.
DIMSUB	Scans the subscript portion of a statement that is dimensioning an array.
EOSR	Processes the end of statement.
ERROR	Enters error intermediate text for errors that cause termination of the processing of that statement.
EQUIV	Processes EQUIVALENCE statements.
EXTERN	Processes EXTERNAL statements.
FORMAT	Processes FORMAT statements.
FUNCT	Processes the header card image for a FUNCTION.
GETWD	Obtains a word or element in a statement and gets a new card image, if necessary. Prints the card if SOURCE option requested. May use PRMBLD.
INTGER/REAL/DOUBLE	Processes INTEGER, REAL, and DOUBLE PRECISION statements.
LABLU	Enters only statement number information into the overflow table. Uses LABTLU.
LABTLU	Enters all information into the overflow table.
LITCON	Processes literals.
LOADE	Performs end-of-phase processing and passes control to Phase 10E.
PRMBLD	Performs all operations associated with I/O interfacing and buffer switching.
PUTBTXT	Puts COMMON and EQUIVALENCE text into SYSUT2 text buffers.
PUTX	Puts entries into the SYSUT1 text buffers.
RCOMA	Enables skipping of redundant commas in a parameter list.
START	Performs initial phase housekeeping.
SUBRUT	Processes the header card for a SUBROUTINE.
SYMTLU	Enters symbols and/or units into the dictionary.
WARN/ERRET	Enters warning and error intermediate text for error and warning conditions that permit the continuation of the processing of the statement.

Chart 06. Phase 10E (IEJFJAA0) Overall Logic Diagram

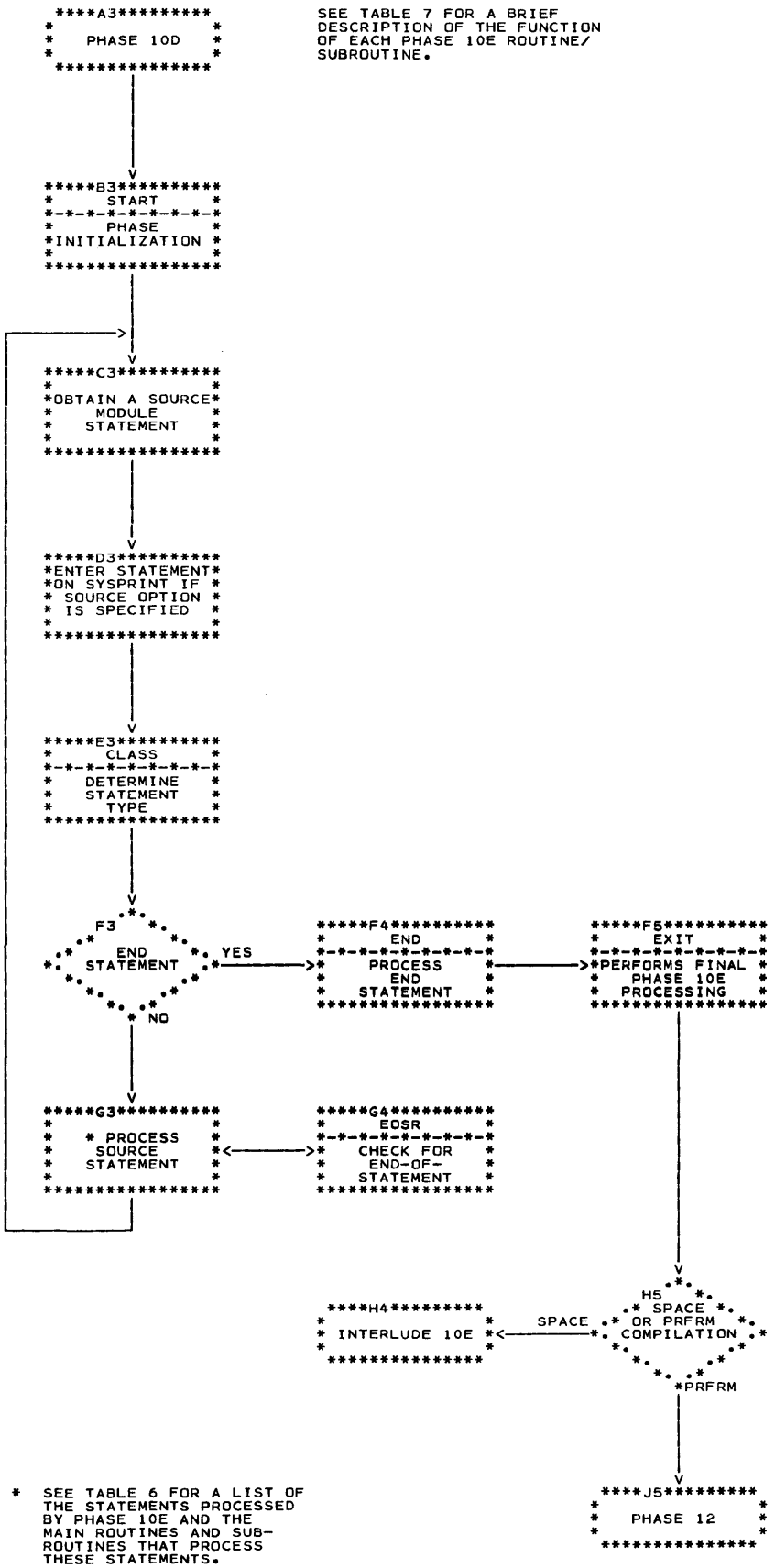


Table 6. Phase 10E Statement Processing

Statement Type	Main Processing Routine		Main Subroutines Used ++
ARITHMETIC	ARITH	* **	CSORN, PUTX, GETWD, SUBS (ARITH may pass control to ASF, DO, and GO)
SF	ASF	* **	CSORN, GETWD
CALL	CALL	* **	PUTX, GETWD, CSORN (exits to ARITH)
DO	DO	* **	ARITH, CSORN, GETWD, LABLU, PUTX
GO TO	GO	* **	ARITH, GETWD, LABLU, PUTX, CSORN, WARN/ERRET
COMP GO TO	GO	* **	
IF	SUBIF	* **	GO, PUTX (exits to ARITH)
READ	READ/WRITE	* **	GETWD, CSORN, PUTX, LABLU (exits to ARITH)
WRITE	READ/WRITE	* **	
FORMAT	FORMAT	*	GETWD, WARN/ERRET, PUTX
CONT	CONT/RETURN	*	GETWD, WARN/ERRET, PUTX
RETURN	CONT/RETURN	*	
STOP	STOP/PAUSE	*	GETWD, PUTX (exits to CLASS)
PAUSE	STOP/PAUSE	*	
BACKSPACE	BKSP/	* **	
REWIND	REWIND/	* **	CSORN, GETWD, PUTX
ENDFILE	ENDFIL	* **	

* Text is created when processing this statement.

** Table entries may be prepared when processing this statement.

++ All routines except FORMAT and CONT/RETURN use ERROR as an error exit for errors that cause termination of the statement processing.

Table 7. Phase 10E Main Routine/Subroutine Directory

Routine/Subroutine	Function
ARITH	Processes arithmetic statements. May use SUBS.
ASF	Processes the parameter list of a statement function.
BKSP/REWIND/ENDFIL	Processes the BACKSPACE, REWIND, and ENDFILE statements.
CALL	Processes the name of a CALL statement.
CLASS	Determines which routine will process the statement type.
CONT/RETURN	Processes CONTINUE and RETURN statements.
CSORN	Processes names, constants, data set reference numbers, and DO parameters. May use LITCON and SYMTLU.
DO	Processes the DO statement and implied DOs.
END	Processes the END statement.
EOSR	Processes the end of the statement.
ERROR	Enters error text into the intermediate text and terminates the processing of current statement.
EXIT	Performs end-of-phase processing.
FORMAT	Processes FORMAT statements.
GETWD	Obtains a word or element in a statement and gets a new card image, if necessary. Prints the card if SOURCE option is requested. May use PRMBLD.
GO	Processes the statement number branched to by an IF, GO TO, or computed GO TO statement.
LABLU	Enters only statement number information into the overflow table. Uses LABTLU.
LABTLU	Enters all information into the overflow table.
LITCON	Processes literals.
PRMBLD	Performs all operations associated with I/O interfacing and buffer switching.
PUTX	Puts entries into the intermediate text buffers.
READ/WRITE	Processes the portion of the statement preceding the I/O list.
START	Performs Phase 10E initialization.
STOP/PAUSE	Processes the STOP and PAUSE statements.
SUBIF	Begins the IF statement processing.
SUBS	Processes subscript variables.
SYMTLU	Enters symbols and/or units into the dictionary.
WARN/ERRET	Processes warning and error conditions that do not prevent completion of the processing of the current statement.

Chart 07. Phase 12 (IEJFLAA0) Overall Logic Diagram

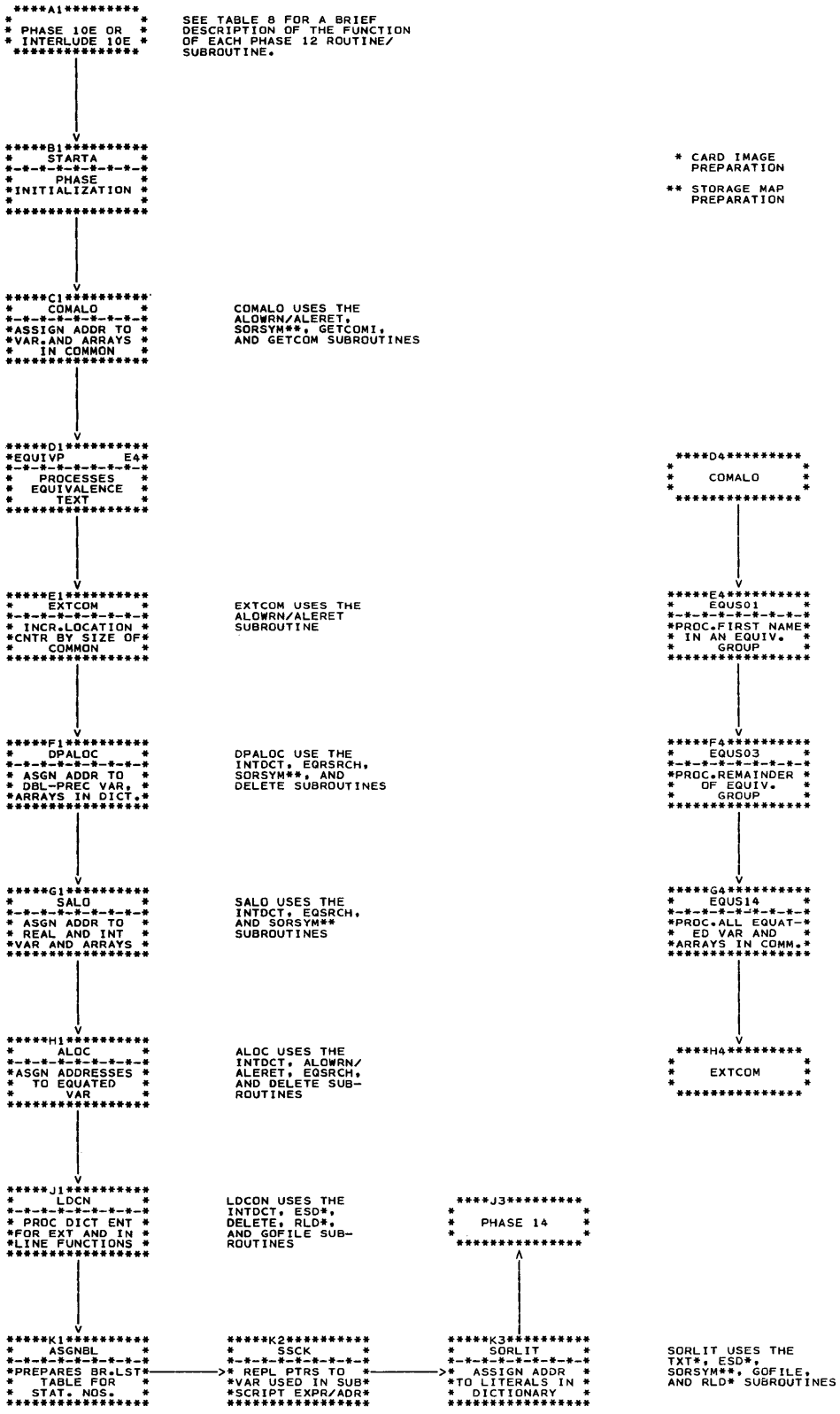


Table 8. Phase 12 Main Routine/Subroutine Directory

Routine/Subroutine	Function
ALOC	Assigns addresses to all equated variables.
ALCWRN/ALERET	Processes the error and warning conditions detected in Phase 12.
ASGNBL	Allocates a branch list position for each referenced stmt. no.
COMALO	Assigns addresses for variables or arrays to be placed in COMMON and removes these variables from the appropriate dictionary chain.
DELETE	Removes dictionary entries from chain.
DPALOC	Assigns addresses to all double-precision variables or arrays entered in the dictionary.
EQSRCH	Checks for variables previously equated to a root.
EQUIVP	Performs equivalence processing.
EQUS01	Processes first name in an EQUIVALENCE group.
EQUS03	Processes remainder of EQUIVALENCE group and switches root if necessary.
EQUS14	Processes all equated variables and arrays in COMMON.
ESD	Processes ESD card images.
EXTCOM	Enters size of COMMON in the communication area.
GETCOM/GETEQUIV	Updates COMMON or EQUIVALENCE text pointer and reads in text records when necessary.
GETCOMI	Initializes pointers and I/O parameters for COMMON-EQUIVALENCE text.
GOFILE	Generates card images for data sets SYSLIN and/or SYSPUNCH.
INTDCT	Retrieves entries from the dictionary.
LDCN	Processes dictionary entries for functions and external references. Also prepares ESD section definition card images for the object module and COMMON areas.
RENTER/ENTR	Enters variables in the EQUIVALENCE table either as a root or as an equated variable.
RLD	Processes RLD card images.
SALO	Assigns addresses to real and integer variables and arrays.
SORLIT	Assigns addresses and generates text card images for all literals; performs the final processing of the phase.
SORSYM	Arranges and prints the storage map for all arrays, constants, and external references assigned addresses by Phase 12.
SSCK	Replaces pointers to variables used in subscript expressions with addresses assigned by Phase 12.
STARTA	Initializes Phase 12.
SWROOT	Changes a root previously entered.
TXT	Processes text card images.

Chart 08. Phase 14 (IEJFNAA0) Overall Logic Diagram

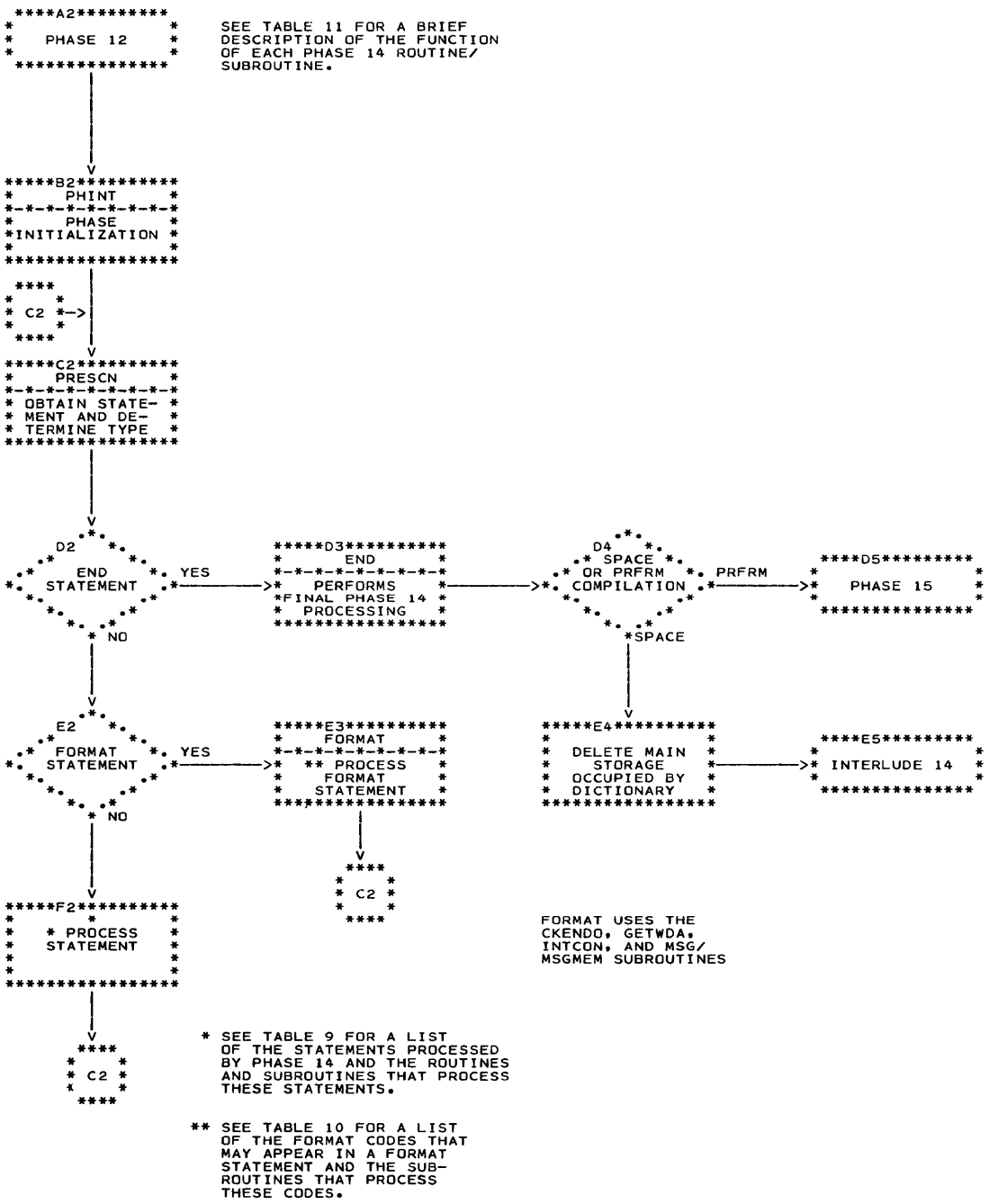


Table 9. Phase 14 Statement Processing (FORMAT Statements Excluded)

Statement Type	Main Processing Routine	Main Subroutines Used
FORMAT	FORMAT	See Table 10
WRITE	READWR	UNITCK, ERROR, MSGMEM
READ	READ	
SUBROUTINE	SUBFUN	RDPOTA*, MSGMEM, RPTRB
FUNCTION	SUBFUN	
CONTINUE	SKIP	MSGMEM
BACKSPACE	BSPREF	UNITCK, MSGMEM
REWIND	BSPREF	
ENDFILE	BSPREF	
DO	DO	CKENDO, ERROR, MSGMEM, RDPOTA*
STATEMENT NUMBER	LABEL	None
SF	ASF	PASSON, CEM, RPTRB
RETURN	RETURN	CKENDO, MSGMEM, SKIP
STOP	STOP	CKENDO, SKIP
PAUSE	PAUSE	CKENDO, SKIP, RDPOTA*
INVALID	INVOP	None
ERROR	ERWNEM	None
WARNING	ERWNEM	
END MARK	MSG	None
IF	PASSON	CEM
ARITH	PASSON	
CALL	PASSON	
GO TO	PASSON	
COMP GO TO	CGOTO	CKENDO, RDPOTA, MSG, MSGMEM

* Replacement of dictionary pointers

Table 10. Phase 14 FORMAT Statement Processing

Processing the Various FORMAT Codes	
FORMAT Code	Main Subroutine Used
blank	BLANKZ
D	FMDCON
E	FMECON
F	FMFCON
I	FMTINT
A	FMACON
X	FMXCON
P	FSCALE
+	FMPLUS
-	FMINUS
(	LPAREN
/	FSLASH
T	FSUBST
H	FHOLER
'	FQUOTE
,	FCOMMA
)	RPAREN

Table 11. Phase 14 Main Routine/Subroutine Directory

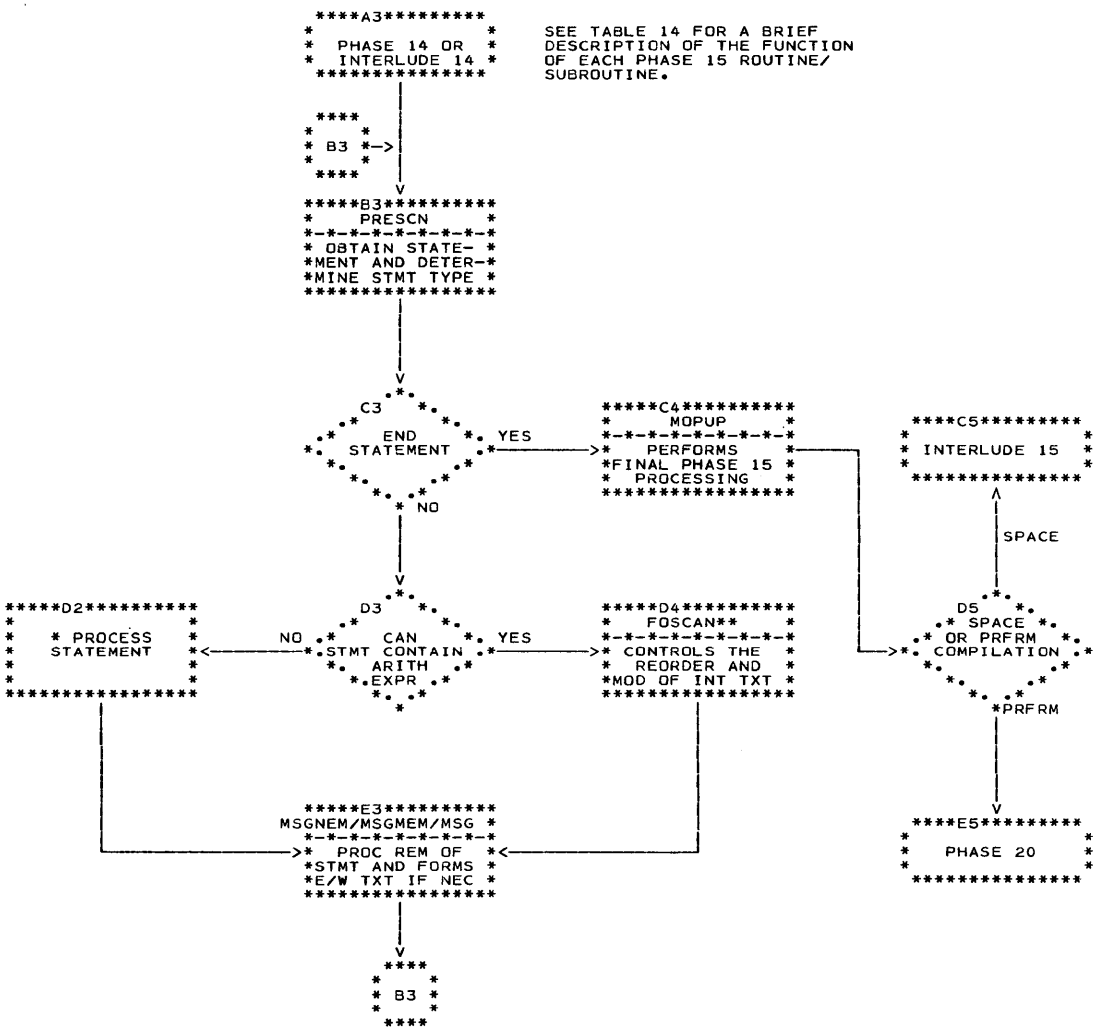
Routine/Subroutine	Function
ASF	Processes the SF definition text.
BLANKZ	Processes any blanks encountered while scanning a FORMAT statement.
BSPREF	Processes BACKSPACE, REWIND, and ENDFILE statement text.
CEM/RDPOTA/RPTRB	Completes text processing for arithmetic, BACKSPACE, REWIND, ENDFILE, GO TO, DO, CALL, IF, PAUSE, and SF definition statements.
CGOTO	Processes text for computed GO TO statements.
CKENDO	Determines if a statement has invalidly ended a DO loop.
DO	Performs diagnostic checks on the DO variable and the DO parameter.
END	Processes END text.
ERROR	Generates intermediate text entries for error conditions detected in Phase 14.
ERWNEM	Processes error and warning text.

(Continued)

Table 11. Phase 14 Main Routine/Subroutine Directory (Continued)

Routine/Subroutine	Function
FCOMMA	Processes any commas found in a FORMAT statement.
FHOLER	Processes the H specification in a FORMAT statement.
FMACON	Processes the A specification in a FORMAT statement.
FMDCON	Processes the D specification in a FORMAT statement.
FMECON	Processes the E specification in a FORMAT statement.
FMFCON	Processes the F specification in a FORMAT statement.
FMINUS	Processes the '-' specification in a FORMAT statement.
FMPIUS	Processes the '+' specification in a FORMAT statement.
FMTINT	Processes the T specification in a FORMAT statement.
FMXCON	Processes the X specification in a FORMAT statement.
FORMAT	Performs and directs some FORMAT processing. May use INTCON.
FQUOTE	Processes the apostrophe specification in a FORMAT statement.
FSCALE	Processes the P specification in a FORMAT statement.
FSLASH	Processes the slash format specification in a FORMAT statement.
FSUBST	Processes the T specification in a FORMAT statement.
GETWDA	Scans FORMAT statements.
INTCON	Converts integer constants to binary and checks their validity.
INVOP	Processes invalid adjective codes.
LABEL	Processes statement number definition text.
LPAREN	Processes left parentheses.
MSG/MSGMEM	Inserts error/warning messages into text and detects end of stmt.
PASSON	Processes CALL, IF, and arithmetic IF statement text.
PAUSE	Processes PAUSE statement text.
PHINT	Performs phase initialization.
PRESCN	Performs phase initialization and controls processing of int. text.
READ/READWR	Processes READ/WRITE text.
RETURN	Processes RETURN statement text.
RPAREN	Processes any right parenthesis occurring in a FORMAT statement.
SKIP	Processes CONTINUE statement text.
STOP	Processes STOP statement text.
SUBFUN	Processes SUBROUTINE and FUNCTION text entries.
UNITCK	Checks validity of symbols used to reference a DSRN.

Chart 09. Phase 15 (IEJFPAA0) Overall Logic Diagram



* SEE TABLE 12 FOR A LIST OF THE NONARITHMETIC STATEMENTS PROCESSED BY PHASE 15 AND THE MAIN ROUTINES AND SUBROUTINES THAT PROCESS THESE STATEMENTS.

** FOSCAN PROCESSES ARITHMETIC, ARITHMETIC IF, STATEMENT FUNCTION AND CALL STATEMENTS. SEE TABLE 13 FOR A LIST OF THE OPERATORS THAT MAY APPEAR IN THE ABOVE STATEMENTS AND THE MAIN ROUTINES AND SUBROUTINES THAT PROCESS THESE OPERATORS.

Table 12. Phase 15 Nonarithmetic Statement Processing

Statement Type or Adjective Cd	Main Processing Routine 1	Main Subroutines Used
COMPUTED GO TO	CGOTO	LAB, CEM
DO	DO	LAB1, CEM
END MARK	MSG	None
ERROR	ERWNEM	None
GOTO	GOTO	LAB, CEM
INVALID	INVOP	ERROR
I/O LIST	BEGIO	MSGMEM
STATEMENT NUMBER	LABEL	ERROR
WARNING	ERWNEM	None
READ/WRITE	DO2	CEM
RETURN/CONTINUE	SKIP	None

[1] Routine MSGNEM/MSGMEM/MSG is entered from all these routines except ERWNEM and LABEL. These two routines return control directly to PRESCN.

Table 13. Phase 15 Arithmetic Operator Processing

Operator	Main Processing Routine	Main Subroutines Used
ADD	ADD	FREER, SAVER*, SYMBOL, MODE, MVSBXX, FINDR, LOADR1
ARGUMENT	COMMA	CKARG, ERROR, WARN, SAVER*, INLIN2, INARG, MSGMEM
CALL FORCING	CALL	MSG
DIVIDE	MULT	SYMBOL, MODE, LOADR1, CHCKGR*, SAVER*, FREER, DIV, MVSBXR, MVSBXX
EQUAL	EQUALS	ERROR, TYPE, MODE, MVSBRX, WARN, MVSBXR, ASFDEF
EXPONENTIATION	EXPON	SYMBOL, MODE, CKARG
FUNCTION(	FUNC	CKARG, INLIN1
ILLEGAL	INVOP	ERROR
LEFT PAREN	LFTPRN	CKARG, ERROR, ARTHIF, WARN, LOADR1
MULTIPLY	MULT	SYMBOL, MODE, MVSBXX, LOADR1, CHCKGR*, FREER
RIGHT PAREN	RTPRN	ERROR
SUBTRACT	ADD	SYMBOL, MODE, MVSBXX, FINDR, LOADR1, FREER, SAVER*
UNARY MINUS	UMINUS	TYPE, FINDR, LOADR1, MVSBRX, INVOP
UNARY PLUS	UPLUS	INVOP

* Specific sections of the SAVER and CHCKGR routines operate upon specific registers (general registers 0, 1, 2, 3; floating point registers 0, 2, 4, 6).

Table 14. Phase 15 Main Routine/Subroutine Directory

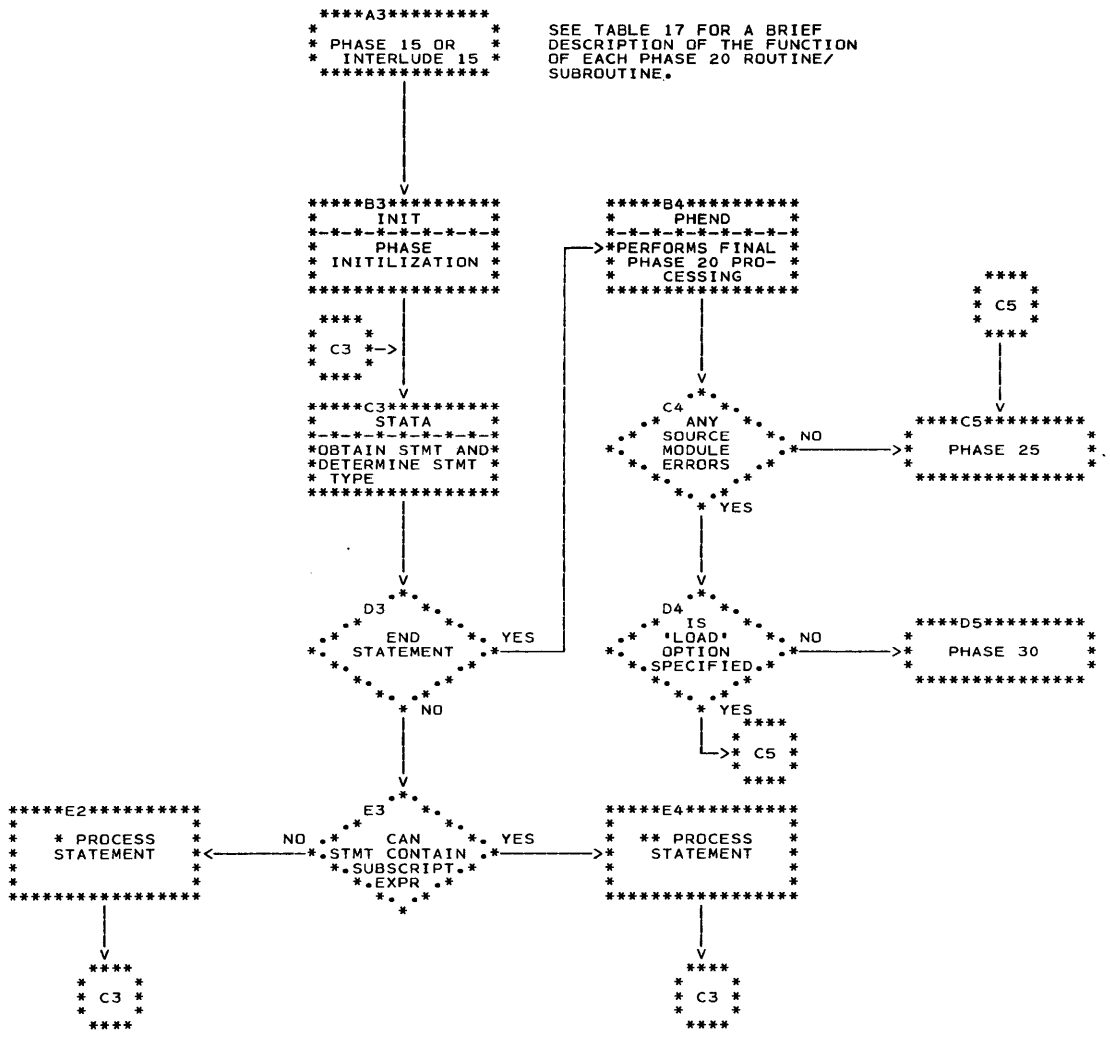
Routine/Subroutine	Function
ADD	Determines register assignment for add, subtract, multiply, and divide operators.
ARTHIF	Processes the statement numbers of an arithmetic IF statement.
ASFDEF	Processes statement function definitions.
BEGIO	Processes the I/O list of READ and WRITE statements.
CALL	Prccesses CALL statements.
CEM	Checks for an end mark.
CHCKGR	Obtains a specific general register for assignment.
CKARG	Checks the argument in an external call for validity, and ensures that the argument has a storage location.
COMMA	Processes the argument lists.
CGOTO	Processes the statement numbers in a computed GO TO statement.
DIV	Processes integer operands of a divide operation.
DO	Processes DO statements.
DO2	Writes out a text word if not an end mark.
END	Determines if the arithmetic IF, arithmetic, and SF statements were processed.
EQUALS	Processes equal adjective code text.
ERROR	Processes error conditions detected in the phase.
ERWNEM	Processes end mark, error, and warning text.
EXPON	Processes exponentiation text.
FINDR	Finds a register and indicates that it is a register.
FOSCAN	Checks the syntax of arithmetic, arithmetic IF, CALL, and SF statements, and orders the arithmetic expression text according to a hierarchy of operators. Uses END.
FREER	Indicates a register is available.
FUNC	Processes one-argument functions.
GOTO	Processes statement numbers referenced by a GO TO statement.
INARG	Processes the argument of an in-line function.
INLIN1	Processes one-argument, in-line functions.
INLIN2	Processes two-argument, in-line functions.
INVOP	Processes invalid adjective codes.
LAB	Checks for illegal statement number references.
LAB1	Checks whether label is defined.

(Continued)

Table 14. Phase 15 Main Routine/Subroutine Directory (Continued)

Routine/Subroutine	Function
LABEL	Checks statement numbers used to indicate the end of a DO loop.
LFTPRN	Process the text for a left parenthesis.
LOADR1	Enters an operand into a specific register.
MODE	Checks the mode of operands and changes them if necessary.
MOPUP	Performs final phase processing for Phase 15.
MSGNEM/MSGMEM/MSG	Processes the remaining text words of a statement and puts out any necessary error, warning, and end do text.
MULT	Aids in processing the operands of multiply and divide instructions.
MVSBXR/MVSBRX	Processes a left operand subscripted variable.
MVSBXX	Processes a left operand subscripted variable if the right operand might also be a subscripted variable.
PRESCN	Determines what statement type is represented in the text and which major routine will process it.
RTPRN	Processes illegal use of right parenthesis as a delimiter.
SAVER	Stores the contents of a specified register into the next available work area space.
SKIP	Processes RETURN and CONTINUE statements.
SYMBOL	Checks the left and right operands of an operator.
TYPE	Checks each symbol used as an operand.
UMINUS	Processes unary minus operations.
UPLUS	Processes unary plus operations.
WARN	Processes warning conditions detected in the phase.

Chart 10. Phase 20 (IEJFRAA0) Overall Logic Diagram



* SEE TABLE 15 FOR A LIST OF- 1) THE STATEMENTS PROCESSED BY PHASE 20 THAT DO NOT CONTAIN SUBSCRIPT EXPRESSIONS, AND- 2) THE MAIN ROUTINE AND SUBROUTINES THAT PROCESS THESE STATEMENTS.

** SEE TABLE 16 FOR A LIST OF- 1) THE STATEMENTS PROCESSED BY PHASE 20 THAT MAY CONTAIN SUBSCRIPT EXPRESSIONS, AND- 2) THE MAIN ROUTINES AND SUB- ROUTINES THAT PROCESS THESE STATEMENTS.

Table 15. Phase 20 Nonsubscript Optimization Processing

Statement Type	Main Processing Routine	Main Subroutines Used
DO	DO	BVLSR, RMVBVL
END DO	ENDDO	None
IMPLIED DO	IOLIST	BVLSR, CALSEQ, RMVBVL, SUBVP
READ	READ	None
STATEMENT NUMBER	LABEL	None

Table 16. Phase 20 Subscript Optimization Processing

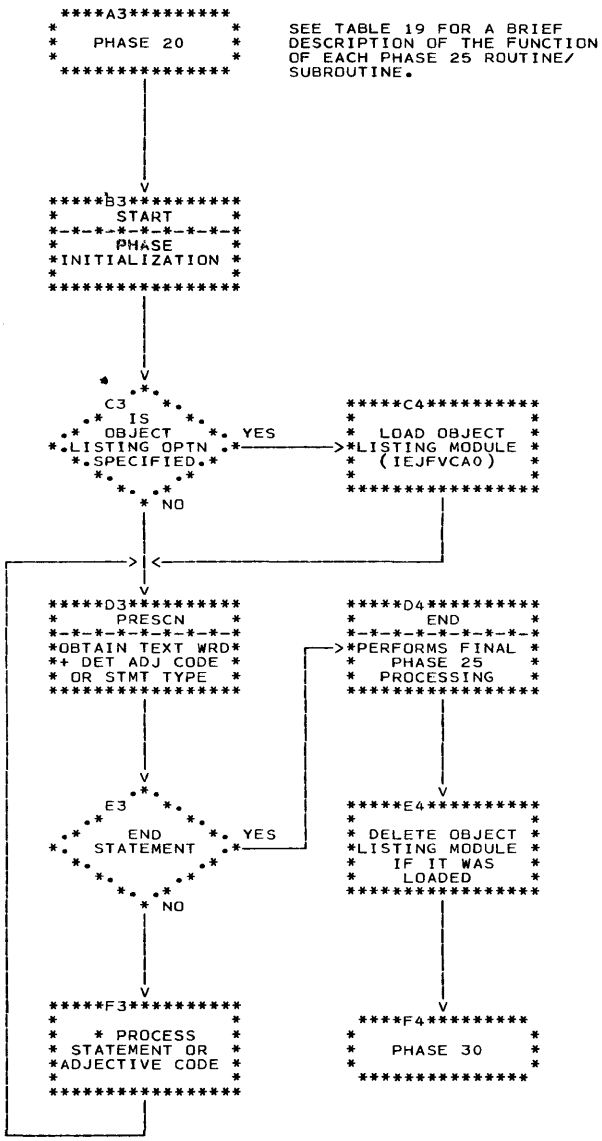
Statement Type	Main Processing Routine	Main Subroutines Used
ARITHMETIC*	ARITH	CALSEQ, CKCOD, RMVBVL, SUBVP
CALL*	IFCALL	BVLSR, CALSEQ, RMVBVL, SUBVP
IF*	IFCALL	None
I/O*	IOLIST	BVLSR, CALSEQ, RMVBVL, SUBVP

* Whenever exponentiation is encountered subroutine ESDRLD processes the exponentiation operands.

Table 17. Phase 20 Main Routine/Subroutine Directory

Routine/Subroutine	Function
ARITH	Optimizes arithmetic statement text.
BVLSR	Enters bound variables on the bound variable list.
CALSEQ	Processes argument lists.
CKCOD	Assigns an area and a constant for use by the IFIX, FLOAT, and DFLOAT in-line functions.
DO	Processes DO statements.
DUMPR	Processes dummy subscripted variables.
ENDDO	Ensures that the end of a DO loop is recognized.
ESDRLD	Generates ESD and RLD card images.
GENGEN	Begins the generation of literals.
IFCALL	Optimizes the arithmetic expression of an arithmetic IF statement or a CALL statement.
INIT	Performs Phase 20 initialization.
IOLIST	Processes DO variables of an implied DO and I/O lists of READ/WRITE statements.
LABEL	Modifies register assignments due to referenced statement numbers.
PHEND	Performs final Phase 20 processing.
READ	Processes external references within a READ statement.
RMVBVL	Removes register assignments from the index mapping table for subscript expressions that involve bound variables.
STATA	Checks the statement type represented by the text and determines the correct Phase 20 processing routine.
SUBVP	Optimizes subscript expressions.

Chart 11. Phase 25 (IEJFVAA0) Overall Logic Diagram



* SEE TABLE 18 FOR A LIST OF THE STATEMENTS AND ADJECTIVE CODES PROCESSED BY PHASE 25 AND THE MAIN ROUTINES AND SUBROUTINES THAT PROCESS THE STATEMENTS OR ADJECTIVE CODES.

Table 18. Phase 25 Statement and Adjective Code Processing

Statement or Operation	Main Processing Routine ****	Main Subroutines Used
AOP	AOP	BASCHK
Arith expressions in approximate instr. form	RXGEN/LM/STM	BASCHK/RROUT, RXOUT
SF DEFINITION	ASFDEF*	LISTOUTB
SF USAGE	ASFUSE	BASCHK/RROUT, RXOUT
BACKSPACE	RDWRT	BASCHK, ARGOUT, GET, RXOUT
CALL	FUNGEN	BASCHK/RROUT
COMPUTED GOTO	CGOTO	BASCHK/RROUT, ARGOUT
DO	DO1	BASCHK, RXOUT
END DO	ENDDO	BASCHK, RXOUT
END FILE	RDWRT	BASCHK, ARGOUT, RXOUT, GET
END I/O LIST	ENDIO	RXOUT
ERROR	IBERR	BASCHK, RROUT
FUNCTION	SUBRUT**	GENBR, GET, RROUT, RXOUT
FUNCTION CALL	FUNGEN	BASCHK/RROUT, RXOUT
GO TO	TRGEN	BASCHK/RROUT, RXOUT
IF	ARITHI	BASCHK/RROUT
IMPLIED DO	DO1	BASCHK, RXOUT, LISTOUTB
I/O LIST ITEM	IOLIST	ARGOUT, BASCHK/RROUT, RXOUT
LABEL	LABEL***	LISTOUT1
LOAD MULTIPLE	LM	BASCHK/RROUT, RXOUT
PAUSE	PAUSE	BASCHK/RROUT, RXOUT
READ/WRITE	RDWRT	BASCHK/RROUT, ARGOUT, RXOUT
RETURN	RETURN	BASCHK/RROUT, RXOUT, LISTOUT1
REWIND	RDWRT	BASCHK, ARGOUT, RXOUT
STOP	STOP	None
STORE MULTIPLE	STM	BASCHK/RROUT, RXOUT
SUBROUTINE	SUBRUT**	GENBR, BASCHK/RROUT, RXOUT
SUBSCRIPT	SAOP	BASCHK/RROUT, RXOUT

* Makes an entry in the statement function and DO branch list table.
** Makes an entry in the epilog table.
*** Makes an entry in the statement number branch list table.
**** All of the above routines return control to the PRESCN routine to begin the processing of the next text word.

Table 19. Phase 25 Main Routine/Subroutine Directory

Routine/Subroutine	Function
AOP	Processes subscript text when the entire subscript expression need not be calculated.
ARGOUT	Inserts addresses for arguments into the object module.
ARITHI	Processes arithmetic IF statements.
ASFDEF	Processes the first text word of a statement function definition.
ASFUSE	Generates instructions to use a statement function at object time.
BASCHK/RROUT, RXOUT	Generates RX and RR format instructions.
CGOTO	Processes computed GO TO statement text.
DO1	Begins processing of the DO statement text.
END	Performs the final Phase 25 processing.
ENDDO	Generates instructions to end a DO loop.
ENDIO	Processes the end I/O text.
FUNGEN/IBERR	Processes in-line and library function calls.
GENBR	Makes entries to the branch list tables.
GET	Obtains intermediate text words.
IOLIST	Processes the I/O list substatement text.
LABEL	Processes statement number definition text entries.
LISTOUTB/LISTOUT1	Generates branch list text.
PRESCN	Determines which routine will process a particular portion of intermediate text.
RDWRT	Processes READ, WRITE, BACKSPACE, REWIND, and ENDFILE statements.
RETURN	Processes RETURN statement text.
RXGEN/LM/STM	Processes intermediate text entries with adjective codes between 25 and 8F (hexadecimal).
SAOP	Processes subscript text when the entire subscript ordering factor must be calculated.
START	Performs phase initialization.
STOP/PAUSE	Generates instructions for the STOP and PAUSE statement text.
SUBRUT	Processes FUNCTION and SUBROUTINE header card text.
TRGEN	Generates branching instructions for GO TO statements.

Chart 12. Phase 30 (IEJFXAA0) Overall Logic Diagram

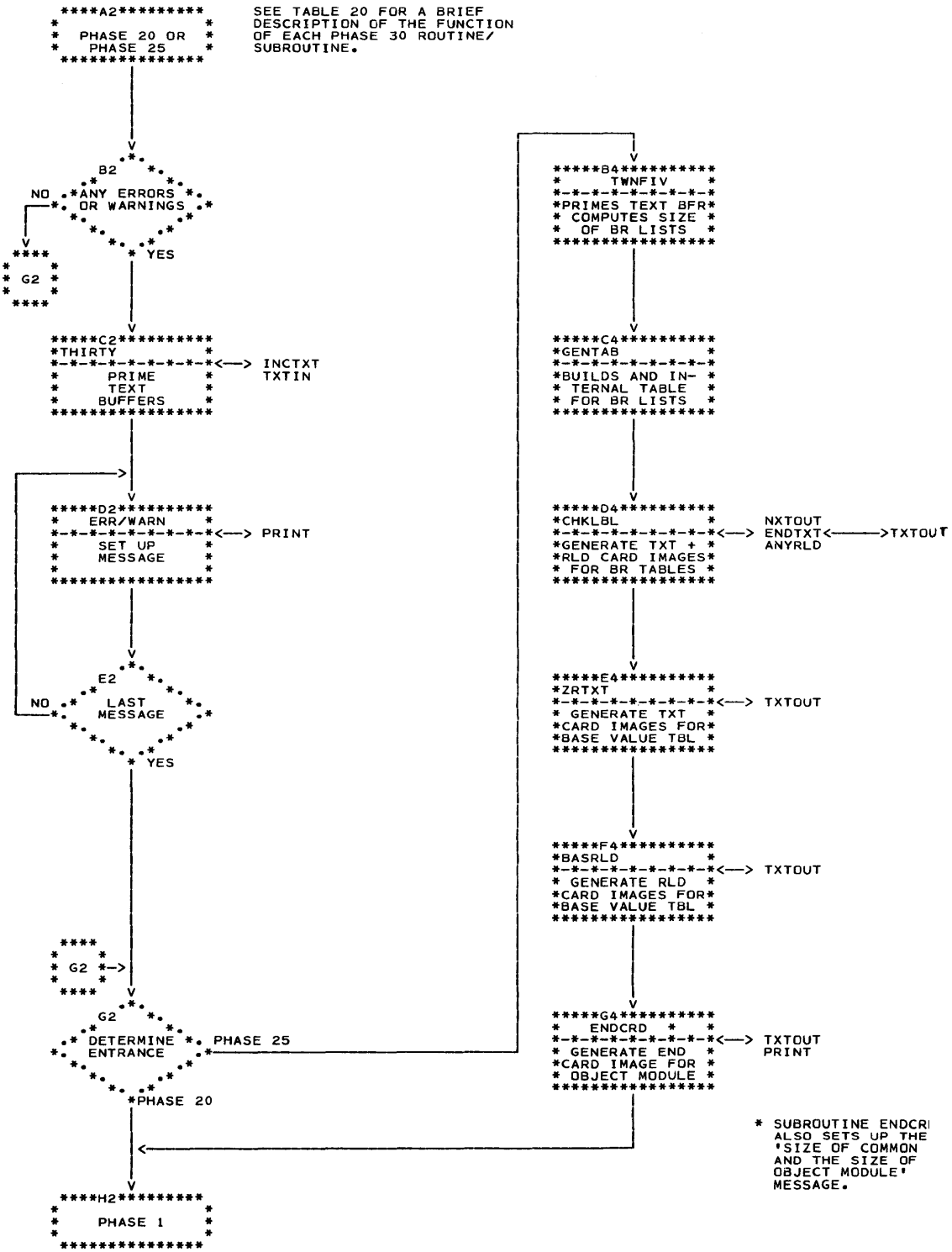


Table 20. Phase 30 Main Routine/Subroutine Directory

Routine/Subroutine	Function
ANYRLD	Generates RLD card images for branch list tables.
BASRLD	Generates RLD card images for base value table.
CHKIBL	Controls generation of TXT and RLD card images for branch lists.
ENDCRD	Generates END card image for object module.
ENDTXT	Switches input/output buffers.
EOJ	Sets up 'SIZE OF COMMON' and 'SIZE OF PROGRAM' message.
ERR/WARN	Sets up error and warning messages.
GENTAB	Builds an internal table for branch list tables.
INCTXT	Increments intermediate text pointer.
NXTOUT	Generates TXT card images for branch list tables.
PRINT	Interfaces with control program to print messages.
THIRTY	Primes input text buffers.
TWNFIV	Primes input text buffers.
TXTIN	Reads intermediate text.
TXTOUT	Outputs card images on SYSLIN and/or SYSPUNCH data sets.
ZRTXT	Generates TXT card images for base value table.

The manipulation of the data control blocks for the data sets required by the compiler depends on whether a SPACE or a PRFRM compilation is being performed. For SPACE compilations, there is more data control block manipulation because of main storage limitations. (The main storage required to contain all the BSAM routines and the control blocks for I/O operations may not be available or may be restricted from the compiler by the value specified in the SIZE option.) For PRFRM compilations, the availability of main storage is not a limitation. Therefore, less data control block manipulation is required.

FOR SPACE COMPILATIONS

For a SPACE compilation, Phase 1 initially opens only the data control blocks for the data sets used by Phases 7, 10D, and 10E (SYSIN, SYSUT1, SYSUT2, SYSPRINT). For the remainder of the compilation, the data control blocks are opened by the interludes only when their corresponding data sets are to be used by a specific compiler component. Each interlude first closes all the data control blocks and then opens only those that are to be used. This process decreases the size of the resident BSAM routines and provides the compiler with the additional main storage necessary for compilation.

Figure 12 illustrates the manipulation of data control blocks for SPACE compilations. OPEN indicates that the data control block is opened during the execution of a compiler component. CLOSE indicates that the data control block is closed during execution of a compiler component. TCLOSE indicates that the corresponding data set is repositioned from the end of the data set to the beginning of the data set for subsequent reading or writing. IN, OUT, INOUT, and OUTIN indicate that the corresponding data set is used for initial or intermediate compiler input, for intermediate or final compiler output, for input followed by output, and for output followed by input. READ indicates that the corresponding data set is read from during execution of a compiler component. WRITE indicates that the corresponding data set is written onto during execution of a compiler component.

For a batch compilation (i.e., more than one source module), the SYSPRINT, SYSLIN, and SYSPUNCH data sets are manipulated so that each data set contains the output for the entire compilation (i.e., for all the source modules). However, if the SYSOUT parameter is used on the DD statements associated with SYSPRINT, SYSLIN, and SYSPUNCH; a new data set is created for the output of each of the compiled source modules.

FOR PRFRM COMPILATIONS

For PRFRM compilations, Phase 1 initially opens the data control blocks for all the data sets required by the compiler. Because all the required data control blocks are opened initially, the compiler can bypass the execution of Interludes 10E, 14, and 15. Bypassing the execution of the interludes reduces data control block manipulation and phase-to-phase transition time; therefore, compilation time is also reduced.

Figure 13 illustrates the manipulation of data control blocks for PRFRM compilations. OPEN indicates that the data control block is opened during execution of a compiler component. CLOSE indicates that the data control block is closed during execution of a compiler component. TCLOSE indicates that the corresponding data set is repositioned from the end of the data set to the beginning of the data set for subsequent reading or writing. IN, OUT, and OUTIN indicate that the corresponding data set is used for initial compiler input, for intermediate or final compiler output, and for output followed by input. READ indicates that the corresponding data set is read from during execution of a compiler component. WRITE indicates that the corresponding data set is written onto during execution of a compiler component.

Compiler Component	DCB for SYSIN	DCB for SYSUT1	DCB for SYSUT2	DCB for SYSPRINT	DCB for SYSLIN *	DCB for SYSPUNCH **
Phase 1 (initial entry)	OPEN IN	OPEN OUT	OPEN OUT	OPEN OUT		
Phase 7	READ			WRITE		
Phase 10D	READ	WRITE	WRITE	WRITE		
Phase 10E	READ	WRITE		WRITE		
Interlude 10E	CLOSE	CLOSE OPEN IN	CLOSE OPEN INOUT	CLOSE OPEN OUT	OPEN OUT	OPEN OUT
Phase 12			READ TCLOSE	WRITE	WRITE	WRITE
Phase 14		READ	WRITE		WRITE	WRITE
Interlude 14		CLOSE OPEN OUT	CLOSE OPEN IN	CLOSE	CLOSE	CLOSE
Phase 15		WRITE	READ			
Interlude 15		CLOSE OPEN INOUT	CLOSE OPEN OUTIN	OPEN OUT	OPEN OUT	OPEN OUT
Phase 20		READ TCLOSE	WRITE TCLOSE	WRITE	WRITE	WRITE
Phase 25		WRITE TCLOSE	READ TCLOSE	WRITE	WRITE	WRITE
Phase 30		READ TCLOSE	READ TCLOSE	WRITE	WRITE	WRITE
Phase 1 (subsequent entries)	OPEN IN	CLOSE OPEN OUT	CLOSE OPEN OUT	CLOSE OPEN OUT	CLOSE	CLOSE
Phase 1 (final entry)	CLOSE	CLOSE	CLOSE	CLOSE	CLOSE	CLOSE

* SYSLIN is used only if the LOAD option is specified.
** SYSPUNCH is used only if the DECK option is specified.

Figure 12. Data Control Block Manipulation for SPACE Compilations

Compiler Component	DCB for SYSIN	DCB for SYSUT1	DCB for SYSUT2	DCB for SYSPRINT	DCB for SYSLIN *	DCB for SYSPUNCH **
Phase 1 (initial entry)	OPEN IN	OPEN OUTIN	OPEN OUTIN	OPEN OUT	OPEN OUT	OPEN OUT
Phase 7	READ			WRITE		
Phase 10D	READ	WRITE	WRITE	WRITE		
Phase 10E	READ	WRITE TCLOSE	TCLOSE	WRITE		
Interlude 10E (not executed)						
Phase 12			READ TCLOSE	WRITE	WRITE	WRITE
Phase 14		READ TCLOSE	WRITE TCLOSE		WRITE	WRITE
Interlude 14 (not executed)						
Phase 15		WRITE TCLOSE	READ TCLOSE			
Interlude 15 (not executed)						
Phase 20		READ TCLOSE	WRITE TCLOSE	WRITE	WRITE	WRITE
Phase 25		WRITE TCLOSE	READ TCLOSE	WRITE	WRITE	WRITE
Phase 30		READ TCLOSE	READ TCLOSE	WRITE	WRITE	WRITE
Phase 1 (restart condition)	CLOSE OPEN IN	CLOSE OPEN OUT	CLOSE OPEN OUT	CLOSE OPEN OUT	CLOSE	CLOSE
Phase 1 (previous compilation - PRFRM)						
Phase 1 (final entry)	CLOSE	CLOSE	CLOSE	CLOSE	CLOSE	CLOSE

* SYSLIN is used only if the LOAD option is specified.
** SYSPUNCH is used only if the DECK option is specified.

Figure 13. Data Control Block Manipulation for PRFRM Compilations

APPENDIX B: TABLES USED BY PHASE LOAD MODULES

During a compilation, the compiler uses the following tables:

- Allocation table.
- Routine displacement tables.
- EQUIVALENCE table.
- Forcing value table.
- Operations table.
- Subscript table.
- Index mapping table.
- Epilog table.
- Message length table.
- Message address table.
- Message text table.

Some tables are actual segments of the phase load modules; others are created during the compilation. Each table is used only by the phase that contains it (as a part of the phase load module) or creates it. The following discussions describe the use and format of each table.

ALLOCATION TABLE

The allocation table is a part of the Phase 7 load module. It is used to allocate the amount of main storage obtained among buffer areas and resident tables. An entry in the allocation table has the form shown in Figure 14.

ROUTINE DISPLACEMENT TABLES

The routine displacement tables for reserved word processing routines are parts of the Phase 10D and Phase 10E load modules. Reserved words are those that indicate a specific FORTRAN statement. The Phase 10D and Phase 10E routine displacement tables are identical in structure and in purpose (locating the processing routine for a given reserved word). The Phase 10D table aids in the location of reserved word routines for declarative statements; the Phase 10E table aids in the location of reserved word routines for executable statements.

Each reserved word causes an entry to be made in the dictionary by Phase 7 (refer to Appendix C). The address field of these entries contains a displacement, used as an indexing value, relative to the start of the appropriate routine displacement table. This index is used to obtain the actual displacement, relative to a base register, of a specific reserved word routine located within the Phase 10D or Phase 10E load module. The effective address of the desired reserved word routine is obtained, by Phase 10D or Phase 10E, by adding this displacement to the value in the base register.

Figures 15 and 16 illustrate the format of the routine displacement tables.

Design Point	Available Storage Over 15360	Storage Used for Dictionary and Overflow Table	Storage Used for the four Internal Text Buffers
200K	189440	65536	4x(3624)
108K	95232	65536	4x(3624)
44K	29696	20326	4x(3000)
15K	0	2216	4x(104)

The design point may be 15, 44, 108, or 200 K (K = 1024 bytes). The remaining fields indicate amounts of storage in bytes. If the amount of main storage available is not at a design point, simple interpolation is performed to divide storage appropriately among buffer areas and resident tables.

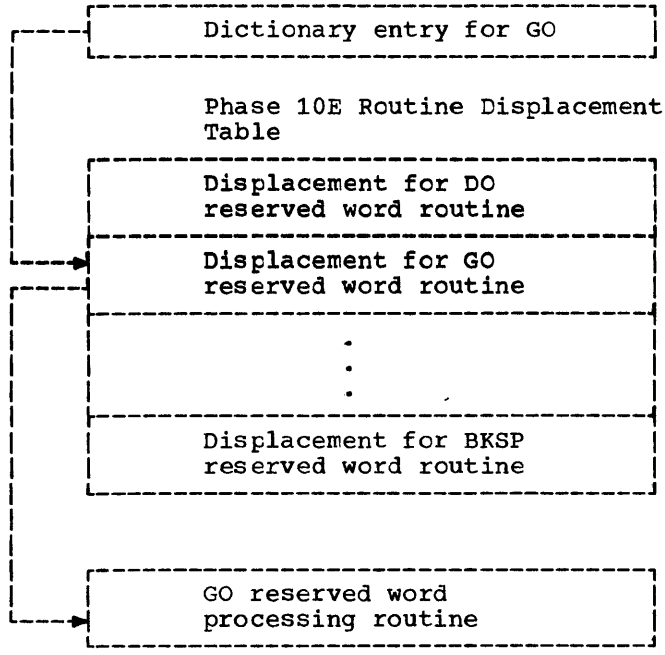
Figure 14. Allocation Table Entry Format

Displacement from base register value of REAL reserved word routine
Displacement from base register value of COMMON reserved word routine
Displacement from base register value of FORMAT reserved word routine
Displacement from base register value of DOUBLE reserved word routine
Displacement from base register value of INTGER reserved word routine
Displacement from base register value of EXTERN reserved word routine
Displacement from base register value of FUNCT reserved word routine
Displacement from base register value of DIM reserved word routine
Displacement from base register value of SUBRUT reserved word routine
Displacement from base register value of EQUIV reserved word routine

2 bytes

Figure 15. Phase 10D Routine Displacement Table Format

The following example illustrates how the GO reserved word routine is located.



Displacement from base register value of DO reserved word routine
Displacement from base register value of GO reserved word routine
Displacement from base register value of FORMAT reserved word routine
Displacement from base register value of IF reserved word routine
Displacement from base register value of END reserved word routine
Displacement from base register value of CALL reserved word routine
Displacement from base register value of GOTO reserved word routine
Displacement from base register value of READ reserved word routine
Displacement from base register value of STOP reserved word routine
Displacement from base register value of PAUSE reserved word routine
Displacement from base register value of WRITE reserved word routine
Displacement from base register value of RETURN reserved word routine
Displacement from base register value of REWIND reserved word routine
Displacement from base register value of ENDFIL reserved word routine
Displacement from base register value of CONT reserved word routine
Displacement from base register value of BKSP reserved word routine

2 bytes

Figure 16. Phase 10E Routine Displacement Table Format

EQUIVALENCE TABLE

The EQUIVALENCE table is constructed by Phase 12 for use by the Phase 12 storage allocation routines, which assign addresses to equated variables. This table is a serial list in which each member follows the preceding one.

The format of a typical entry in the EQUIVALENCE table is shown in Figure 17.

p(variable) or p(array)	p(root)	displacement or address in COMMON	size
2 bytes	2 bytes	2 bytes	2 bytes

Figure 17. EQUIVALENCE Table Entry Format

Each field in an entry is two bytes in length. The first field contains a pointer to the entry for the variable or array in the dictionary. The second field contains a pointer to the dictionary entry for the root to which the variable or array is equated. (If the variable or array is the root of the EQUIVALENCE group, the first two fields contain the same pointer.) The third field contains the displacement or address assigned to the variable or array in COMMON. (The addresses for variables and arrays are assigned before this table is constructed.) The fourth field is the size, in bytes, of the EQUIVALENCE group or class.

The maximum number of entries in the EQUIVALENCE table is the larger of:

- 100, or
- The largest unused segment of the dictionary and overflow table divided by eight (if this segment exceeds 800 bytes).

For example, if the compiler allocates 5500 bytes to the dictionary and the overflow table, and 3100 bytes are used, then the maximum number of entries in the EQUIVALENCE table is:

(5500 - 3100)/8 = 2400/8 = 300

Adjective Code	Left Forcing Value	Address of Associated Routine	Right Forcing Value
(	64	a(LFTPRN)	01
)	00	a(RTPRN)	69
=	70	a(EQUALS)	70
,	49	a(COMMA)	48
n	80	never forced out	01
+	09	a(ADD)	09
-	09	a(ADD)	09
*	05	a(MULT)	05
/	05	a(MULT)	05
**	04	a(EXPON)	03
F(	64	a(FUNC)	01
unary -	05	a(UMINUS)	01
end mark	00	never forced out	80
unary +	05	a(UPLUS)	01
SF Forcing	72	a(END)	70
ARITH Forcing	72	a(END)	70
CALL Forcing	72	a(CALL)	70
IF Forcing	72	a(END)	70
1 byte	1 byte	2 bytes	1 byte

Figure 18. Forcing Value Table

FORCING VALUE TABLE

The forcing value table is not created or altered in any way by the compiler; it is loaded into main storage as a part of the Phase 15 load module. The forcing value table is used by Phase 15 as an aid in the reordering of intermediate text entries in arithmetic expressions. This table defines the relative position of each operator in the hierarchy of operators.

Each entry in the forcing value table is five bytes in length. The forcing value table is illustrated in Figure 18.

OPERATIONS TABLE

The operations table is a temporary storage area (part of the Phase 15 load module) used during the reordering of operations within statements that can contain arithmetic expressions. This table functions as a "pushdown table" (that is, a table in which the top entry is the most recently entered item) for storing intermediate text words that refer to the operation in question. An exception is made for subscript text, which is stored in the subscript table.

The operations table can contain no more than 50 entries. Entries are four bytes in length and are obtained by a pointer to the last entry in the table for the specific statement under consideration. The format of a typical entry in the operations table is shown in Figure 19.

adj code	mode/type field	pointer field
1 byte	1 byte	2 bytes

Figure 19. Operations Table Entry Format

SUBSCRIPT TABLE

The subscript table is a temporary storage area (part of the Phase 15 load module) used for subscript text encountered during the reordering of intermediate text words by Phase 15. This table functions as a "pushdown table" (that is, a table in which the top entry is the most recently entered item) for storing subscript intermediate text words that refer to the operation in question.

The subscript table can contain no more than 38 entries. Entries are eight bytes in length and are obtained by a pointer to the top entry in the table for the specific statement under consideration. The format of a typical entry in the subscript table is shown in Figure 20.

The subscript adjective code indicates to other phases of the compiler that subscript calculation is necessary. The offset is an index used to find the correct element in an array associated with a particular subscript expression. The second word of an entry in the subscript table contains two pointers to information in the overflow table. The first points to the subscript information for the subscripted variable; the second points to the dimension information for the array indicated by the subscripted variable.

INDEX MAPPING TABLE

The index mapping table (part of the Phase 20 load module) is used to aid the implementation of subscript optimization. This table maintains a record of all information pertinent to a subscript expression. Because the computation of any unique subscript expression is placed in a register, the number of entries in the table depends on the number of registers available for this purpose. The initial register assigned to a subscript expression is determined during the initialization process for Phase 20. Each entry in the index mapping table is eight bytes in length. The format of a typical entry in the index mapping table is shown in Figure 21.

The register number field contains the number of the register assigned to the subscript expression. The dimension number field contains the number 1, 2, or 3, depending on the number of dimensions. The status field indicates whether the register referenced by this entry is: (1) unassigned, (2) assigned to a normal subscript expression for indexing computation, or (3) **assigned to the address of a dummy variable. The offset field contains the offset index used to obtain the correct element of the array associated with a particular subscript expression. The last two fields contain pointers to information in the overflow table.**

subscript adjective code	not used by Phase 15	offset	p(subscript)	p(dimension)
1 byte	1 byte	2 bytes	2 bytes	2 bytes

Figure 20. Subscript Table Entry Format

Register number	Number of Dimensions	Status	Offset	p(subscript)	p(dimension)
1 byte	1 byte	2 bytes	2 bytes	2 bytes	

Figure 21. Index Mapping Table Entry Format

EPILOG TABLE

The epilog table is created by Phase 25 when the FUNCTION or SUBROUTINE adjective code is encountered. An entry is made in the epilog table for each variable used as a parameter in the calling program. The instructions generated during Phase 25 for the RETURN entry in the intermediate text reference the epilog table to return the value of variables to the calling program.

Each entry in the epilog table is four bytes in length. The format of a typical entry in the epilog table is shown in Figure 22.

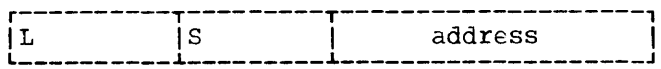


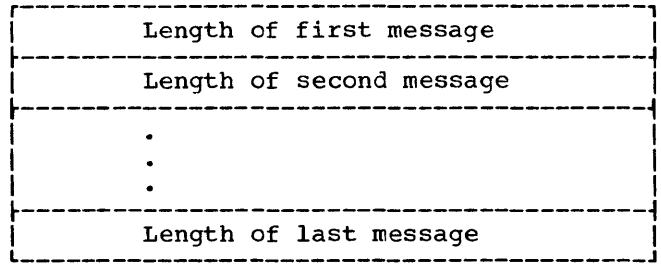
Figure 22. Epilog Table Entry Format

L is the field length of the variable in the subprogram, S is the relative position of the variable in the parameter list of the calling program, and address is the address of the variable in the subprogram.

MESSAGE LENGTH TABLE

The message length table is loaded into main storage as a part of the Phase 30 load module. It contains the lengths of all the messages capable of being generated by the compiler. The length of any message is obtained by using the number corresponding to that message as a displacement from the start of the message length table.

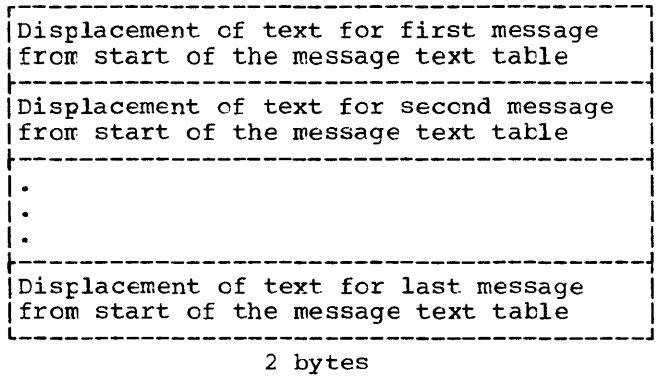
The message length table has the following format:



MESSAGE ADDRESS TABLE

The message address table is loaded into main storage as a part of the Phase 30 load module. It contains the displacements from the start of the message text table of all the messages capable of being generated by the compiler. The displacement of any message is obtained by using the number corresponding to the message multiplied by two as a displacement from the start of the message address table.

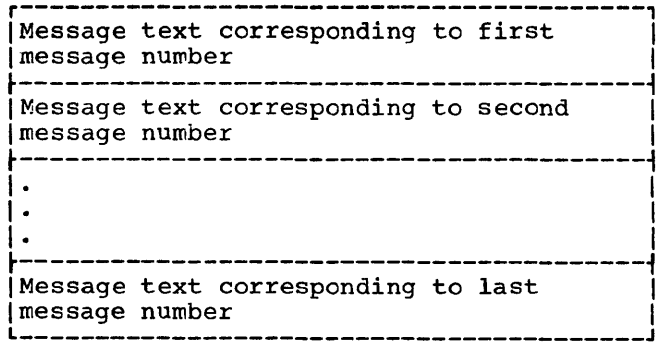
The message address table has the following format:



MESSAGE TEXT TABLE

The message text table is loaded into main storage as a part of the Phase 30 load module. It contains all the messages capable of being generated by the compiler. Each message is obtained by using the displacements contained in the message address table.

The message text table has the following format:



The resident tables of the compiler are:

- The dictionary.
- The overflow table.
- The segment address list (SEGMAL).
- The patch table.
- The blocking table (resident only for PRFRM compilations).
- The BLDL table (resident only for PRFRM compilations).

The dictionary is a reference area containing information about variables, arrays, constants, data set reference numbers, etc., used in the source module. The overflow table contains all dimension, subscript, and statement number information within the source module. SEGMAL is used for main storage allocation within the compiler. The patch table contains information to be used to modify compiler components. The blocking table contains the information necessary for deblocking compiler input and blocking compiler output for PRFRM compilations. The BLDL table contains the information necessary for transferring control from one component of the compiler to the next for PRFRM compilations.

THE DICTIONARY

The dictionary (constructed by Phases 7, 10D, and 10E) is used and modified by Phase 12 in address assignment, and is further used by Phase 14 when addresses from the dictionary replace pointers to the dictionary in the intermediate text entries (refer to Appendix D). For SPACE compilations, Phase 14 frees the dictionary area of storage for use by subsequent phases.

The dictionary is organized as a series of chains related by the dictionary index, which indicates the first entry in each chain. There are 15 chains, used for various entries, as follows:

- Eleven are organized on the basis of length of the symbol being entered (e.g., DO has a length of 2, END has a length of 3, etc.). The first chain is for entries of length 1, the second is for entries of length 2, the third is for entries of length 3, and so on.

  These chains contain entries for reserved words (chains 2-11), in-line functions, variables, and arrays.

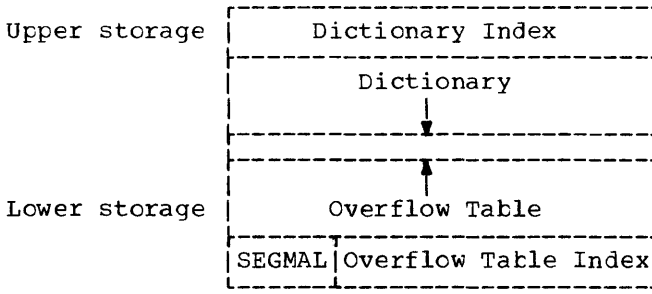
- One chain for real constants.
- One chain for integer constants.
- One chain for integer data set reference numbers.
- One chain for double-precision constants.

Phase 7 Processing

Phase 7 allocates storage for the dictionary, and then enters all reserved words (words that indicate a specific FORTRAN statement) into the dictionary.

Figure 23 illustrates the dictionary after it is constructed by Phase 7.

The dictionary, dictionary index, the overflow table, overflow table index, and SEGMAL are in main storage in the following relative positions.



This order is set up during Phase 7. (Refer to the Phase 7 discussion.)

Phases 10D and 10E Processing

Additions to the dictionary occur as entries are made to the various chains during Phases 10D and 10E processing. To enter an item in the dictionary, the pertinent chain is located via the dictionary index. The chain is searched until the last entry is found. The current end-of-chain indicator is replaced with a pointer to the new entry; the new entry is then marked as the end of the chain.

For example, assume the variable ABC is to be entered in the dictionary. ABC belongs in the third chain of the dictionary (length 3). Using the dictionary index, the first entry of the chain for

length 3 is obtained. Assume that Figure 23 indicates the condition of the dictionary at this time. The chain for length 3 is searched for the last entry (the entry for DIM), which is modified to appear as:

pointer to the entry for ABC	entry for DIM

The entry for ABC appears as:

end of chain	entry for ABC

When the dictionary and overflow table overlap, a message is issued; no new entries are made; and compilation proceeds.

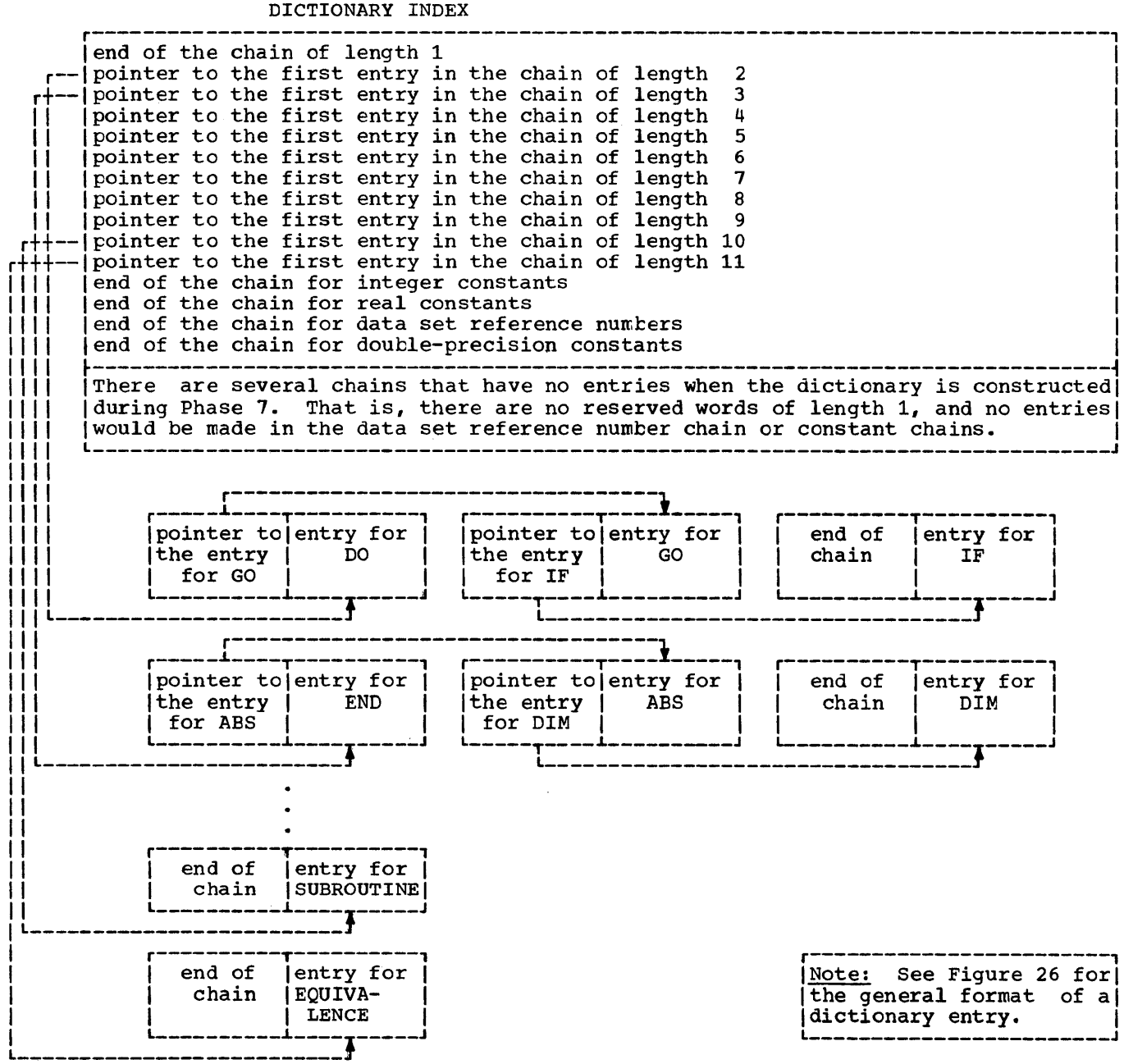


Figure 23. The Dictionary as Constructed by Phase 7

Phase 12 Processing

During the Phase 12 processing, addresses are assigned to the symbols entered in the first six chains of the dictionary. In assigning these addresses, Phase 12 uses the contents of the dictionary entries. The addresses replace: (1) the pointers to following entries in the dictionary, and (2) the end-of-chain indicators. To ensure that the chain is not broken, the chain is continued by modifying the pointer to the entry just assigned an address. Figures 24 and 25 illustrate two cases of the "before" and "after" in removing an entry from a dictionary chain. Figure 24 indicates removal of an entry from the end of the chain. Figure 25 indicates removal of an entry from the middle of the chain.

Phase 14 Processing

During Phase 14 processing, each pointer (in the intermediate text) to a dictionary entry is replaced by the address assigned to the symbol within the dictionary entry. Refer to Appendix D for the modification of the intermediate text.

Dictionary Entry Format

The entries to the dictionary may vary; however, they all have the same general form. Figure 26 indicates this general form.

Chain address field	Usage field	Mode Type field	Image field	Address field	Size field
2 bytes	1 byte	1 byte	1-11 bytes	2 bytes	2 bytes

Figure 26. General Form of a Dictionary Entry

Each field contains specific information as indicated below:

CHAIN ADDRESS FIELD: The chain address field is used to maintain the linkage between the various elements of the chain. It either contains the relative pointer to the next entry or indicates that its associated entry is the last entry in the chain.

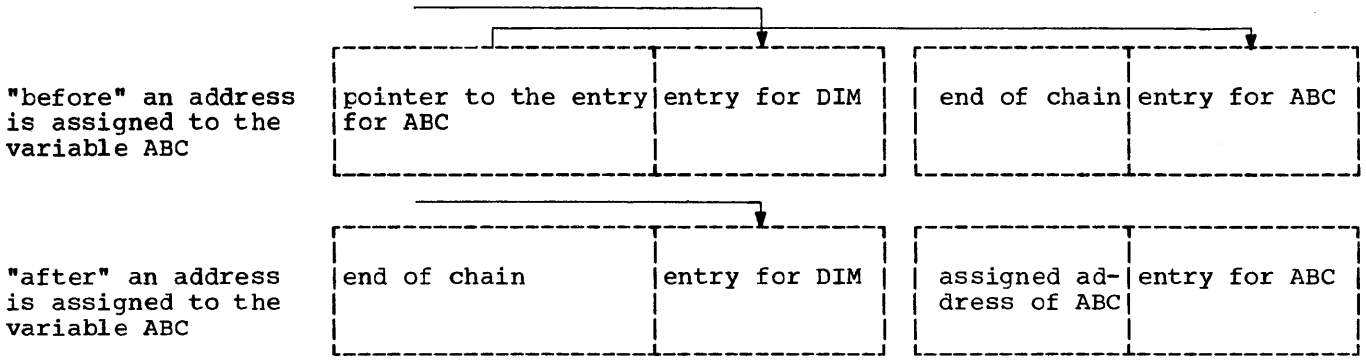


Figure 24. Removing an Entry From the End of a Dictionary Chain

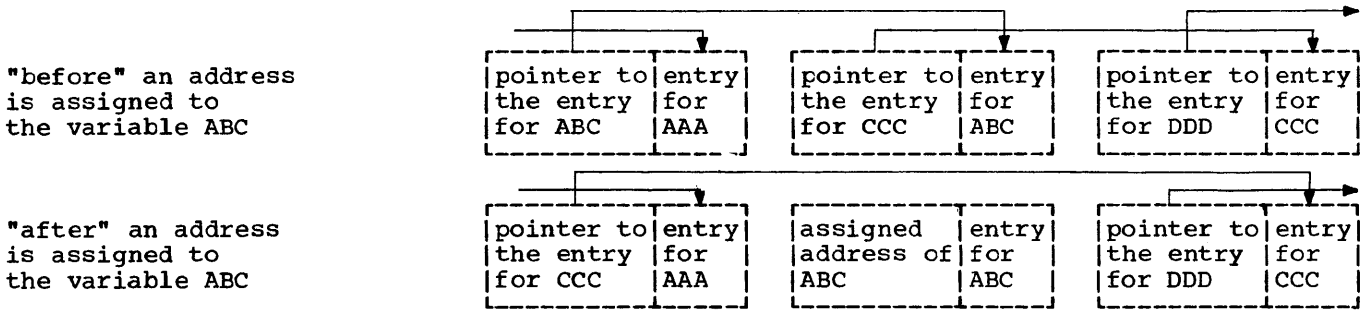


Figure 25. Removing an Entry From the Middle of a Dictionary Chain

USAGE FIELD: The usage field is divided into eight subfields. Each subfield is one bit long and is numbered from 0 through 7, inclusive. Figure 27 indicates the function of each subfield in the usage field.

Usage field subfield	Function of the field
Bit 0	Indicates if the mode of the entry has been defined
Bit 1	Indicates if the type of the entry has been defined
Bit 2	Indicates if the entry is in COMMON
Bit 3	Indicates if the entry is equated
Bit 4	Indicates if the entry is assigned an address
Bit 5	Indicates if this is the entry for the root of an EQUIVALENCE group (see Phase 12)
Bit 6	Indicates if the entry represents double precision
Bit 7	Indicates if the entry is for an in-line function or an external reference.

Figure 27. Function of Each Subfield in the Dictionary Usage Field

MODE/TYPE FIELD: This field is divided into two parts (each four bits long). The first four bits are used to indicate the mode of an entry, while the last four bits are used to indicate the type. For example, a real quantity has the mode code 7; therefore, the mode field for a real is 0111 (the bit configuration for 7). Similarly, a subscripted variable has the type code C; therefore, the type field for a subscripted variable is 1100 (the bit configuration for C). The mode/type field for a real subscripted variable is 01111100. The various mode/type combinations possible are indicated in Figure 28.

IMAGE FIELD: The image field contains the appropriate image of the symbol. The length of the symbol determines the length of the field.

ADDRESS FIELD: The address field is present in dictionary entries for:

- Reserved words -- to indicate the position of the displacement of the processing routine for that reserved word in the Phase 10D or Phase 10E Routine Displacement Table (see Appendix B).
- In-line functions -- to indicate the code value used within the compilation for that in-line function.
- Arrays -- to indicate the displacement within the overflow table of the dimension information for that array.

SIZE FIELD: The size field is present for the dictionary entries that represent arrays. It indicates the size of the array.

All fields are present in each dictionary entry, except the address field and the size field. The fields and the phases that enter information into the fields are indicated in Figure 29.

H\L I\O G\W H\	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0																
1		statement number		unit		*immediate constants			*sub-program	*dummy sub-program						
2					*	*	*	*	*	*	*	*				
3									external					dummy		
4									real				subscript	subscript		
5 integer					gen		status	in	&				subscript	subscript		
6 double precision					generated		mem	-line	library	dummy		dum my	pointer	pointer		dummy
7 real					work	constant	function	function	function	function	variable	variable	variable	variable		dummy
8					area	constant	ion	ion	ion	ion	able	able	able	able	array	array

* Subject to change after Phases 10D and 10E

Figure 28. The Various Mode/Type Combinations

FIELD entries for:	Chain address field	Usage field 0	1	2	3	4	5	6	7	Mode/Type field		Image field	Address field		Size field
Reserved words	7	7	7							7	7	7	7		
In-line functions	7		7							7	7	7	7		
Variables	10D 10E	10D 10E	10D 10E	10D	10D		12	12	10D 10E	10D 10E	10D 10E	10D 10E			
Arrays	10D 10E	10D	10D	10D	10D		12	12	10D 10E	10D 10E	10D 10E	10D 10E	10D		10D
Constants	10D 10E	10D 10E	10D 10E							10E	10E	10D 10E			
Data set reference numbers	10E	10E	10E								10E	10E			

Figure 29. Phases That Enter Information Into Specific Fields of a Dictionary Entry

The overflow table is constructed by Phases 7, 10D, and 10E. The overflow table subscript entries are modified by Phase 12 during address assignment; statement number entries are assigned relative branch list numbers. The overflow table is used by:

- Phase 12 -- to reserve storage for the branch list.
- Phase 20 -- for subscript optimization.
- Phase 25 -- for the construction of object module coding.

Organization of the Overflow Table

The overflow table is organized as a series of chains related by the overflow index. The overflow index indicates the displacement of the first entry in each chain relative to the beginning of the table. There are 11 chains, used for various entries, as follows:

- Three chains are organized for the dimension information of an array; that is, for 1-, 2-, and 3-dimensional arrays.
- Three chains are organized for subscript information; that is, for 1-, 2-, and 3-dimensional subscripts.
- Five chains are organized for statement number information. All statement numbers ending in 0 and 1 are entered in the first chain. The remaining chains handle statement numbers ending in 2 and 3, 4 and 5, 6 and 7, and 8 and 9, respectively.

Construction of the Overflow Table

Phase 7 allocates storage for the overflow table. Because there are no reserved words entered in the overflow table as in the dictionary, only the overflow index is actually constructed. The index contains the end-of-chain indicator for each chain, as no entries exist in any chain at this time. Figure 30 indicates the overflow table as it appears after it is constructed by Phase 7.

Phases 10D and 10E construct all entries to the overflow table. Each entry is entered in an overflow table chain; e.g., assume the 1-dimensional array ARRY1 is the first array entered in Phase 10D. The first overflow index entry is modified to contain:

pointer to the dimension entry for ARRY1

The overflow table entry (in the first array chain) appears as:

end of chain	entry for ARRY1

When the next 1-dimensional array, ARRY2, is entered in the overflow table, the entry for ARRY1 is modified as follows:

pointer to the entry for ARRY2	entry for ARRY1

and the entry for ARRY2 appears as:

end of chain	entry for ARRY2

The entries to other chains are made in like manner during the Phase 10D and the Phase 10E processing.

end of chain for information on 1-dimensional arrays
end of chain for information on 2-dimensional arrays
end of chain for information on 3-dimensional arrays
end of chain for information on 1-dimensional subscripts
end of chain for information on 2-dimensional subscripts
end of chain for information on 3-dimensional subscripts
end of chain for information on statement numbers ending in 0 or 1
end of chain for information on statement numbers ending in 2 or 3
end of chain for information on statement numbers ending in 4 or 5
end of chain for information on statement numbers ending in 6 or 7
end of chain for information on statement numbers ending in 8 or 9

Figure 30. The Overflow Table Index as Constructed by Phase 7

Use of the Overflow Table

Phase 12 modifies the statement number chains when the branch list table for statement numbers (see Appendix F) is prepared initially by Phase 12. The chain field is replaced by a number that indicates the position the statement number has in the branch list table.

Phases 14 and 15 do not use the overflow table.

Phase 20 uses the information about subscripted expressions in performing its function of subscript optimization. This information is obtained via a pointer, in the intermediate text, to the pertinent overflow table entry (in a subscript chain).

Phase 25 uses the branch list table number, assigned by Phase 12, to determine the position of a statement number in the branch table. (Phase 25 can then insert the object-time address, associated with the statement number, in the table.) The number is obtained via a pointer, in the statement number intermediate text entry, to the overflow table.

Overflow Table Entry

An entry in the overflow table has one of three formats:

1. Dimension.
2. Subscript.
3. Statement number.

DIMENSION ENTRY: A dimension entry is formed for each array. An array may be defined as:

- 1-dimensional, e.g., ARRAY (D1).
- 2-dimensional, e.g., ARRAY (D1,D2).
- 3-dimensional, e.g., ARRAY (D1,D2,D3).

One-dimensional arrays are entered in the first dimension chain of the overflow table, 2-dimensional arrays in the second, and 3-dimensional arrays in the third. The formats for the entries of 1-, 2-, and 3-dimensional arrays are indicated in Figure 31.

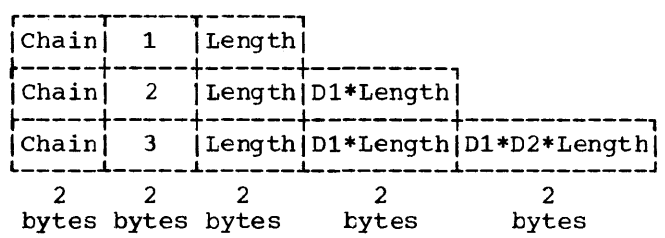
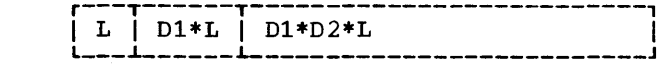


Figure 31. Format of Dimension Information in the Overflow Table

The fields of a dimension entry contain the following information:

- The first field contains the displacement (relative to the beginning of the overflow table) of the next element in the chain.
- The second field is a digit, either 1, 2, or 3, to indicate whether one, two, or three fields will follow. This is the same as the number of dimensions.
- The next field is of the form:



where:

D1*L and D1*D2*L are optional fields depending on the dimension.

I indicates the length of an element in bytes (e.g., 4 for integer or real quantities and 8 for double-precision quantities).

D1 represents the value of the first dimension of the array.

D2 represents the value of the second dimension of the array.

SUBSCRIPT ENTRY: A subscript entry is formed for each subscripted variable. A subscripted variable may be defined as:

- 1-dimensional, e.g., A(I)
- 2-dimensional, e.g., A(I,J)
- 3-dimensional, e.g., A(I,J,K)

One-dimensional subscripts are entered in the first subscript chain of the overflow table, 2-dimensional subscripts in the second, and 3-dimensional subscripts in the third. The formats for the entries of 1-, 2-, and 3-dimensional subscripts are illustrated in Figure 32.

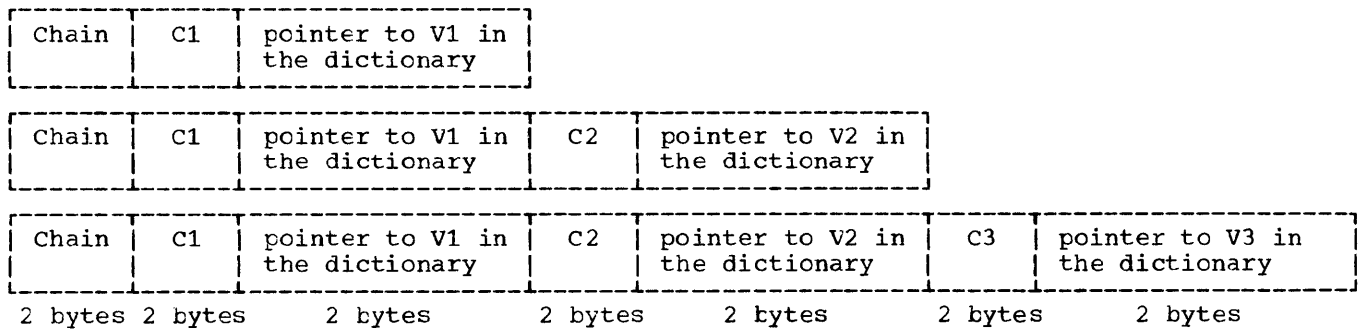


Figure 32. Format of Subscript Information in the Overflow Table

The fields of a subscript entry contain the following information:

- The first field contains the displacement (relative to the beginning of the overflow table) of the next element in the chain.
- The second and third, fourth and fifth, and sixth and seventh fields represent the first, second, and third dimensions of the subscript. The explanation and use of C1, V1, C2, V2, C3, and V3 are given in Appendix E.

STATEMENT NUMBER ENTRY: A statement number entry is constructed for each statement number encountered in the source statements. The format of an entry in the statement number chains is illustrated in Figure 33.

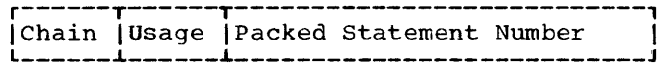


Figure 33. Format of Statement Number Information in the Overflow Table

The fields of a statement number entry contain the following information:

- The first field contains the displacement (relative to the beginning of the overflow table) of the next element in the chain.
- The second field is a usage field where each bit represents the following:

Usage Field Bit	Function of the Field
0	Indicates if the statement number is defined
1	Indicates if the statement number is referenced
2	Indicates if the statement number represents the end of a DO loop
3	Indicates if the statement number represents a specification statement
4	Indicates if the statement number represents a FORMAT statement
5	Indicates if the statement number indicates DO nesting errors
6	Not used
7	Not used

- The third field contains the actual statement number (as it appeared in the source statement) in packed form.

SEGMAL

SEGMAL is constructed by Phase 7 and contains the beginning and ending address of each segment of main storage assigned to the dictionary and overflow table by Phase 7. This main storage is assigned to the compiler as a result of the GETMAIN macro-instruction issued by the compiler during Phase 7. Phases 10D and 10E use SEGMAL as

they enter various items in the dictionary and the overflow table.

Phase 7 Processing

When SEGMAL is constructed by Phase 7, the various segments are put into ascending order; that is, the segment entries of main storage are sorted. Contiguous segments are then combined into a single segment.

The communication area contains information to indicate which segment is currently being used for the overflow table and which is currently being used for the dictionary.

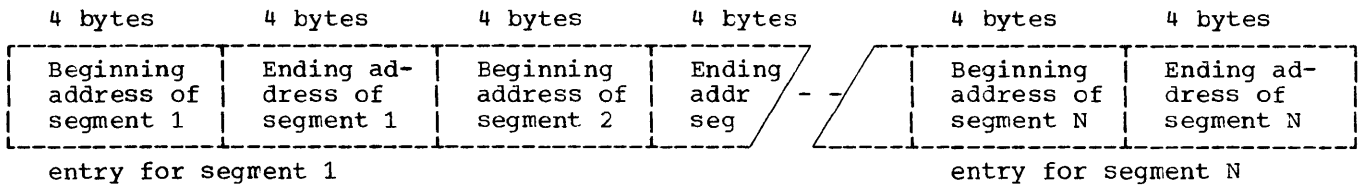
Phases 10D and 10E Processing

Phases 10D and 10E use SEGMAL when new segments of the dictionary and overflow table are required. For SPACE compilations, Phase 14 uses SEGMAL to free the main storage areas allocated to the dictionary.

Format of SEGMAL

SEGMAL has the following form for N segments, where each segment is entered in ascending sequence by address. The entry for each segment consists of the beginning address of the segment and the ending address of the segment. (The storage location containing the ending address of segment N is adjacent to the storage location containing the starting address of the overflow index. The starting address of the overflow index is an entry in the communication area.)

Note: The ending address of segment N minus the beginning address of segment 1 must be less than or equal to 65536.



PATCH TABLE

The patch table (100 bytes) is a part of the interface module. It is used only if the patch facility has been enabled and if patch records precede the source statements of the FORTRAN source module being compiled. The patch table contains a converted form (for internal use) of the information contained in the patch records. The patch table has the following format:

Identifier for first module to be modified	2 bytes
Relative address of first patch for this module	2 bytes
Length (in bytes) of first patch for this module	2 bytes
First patch for this module	Variable
.	.
.	.
.	.
Relative address of last patch for this module	2 bytes
Length (in bytes) of last patch for this module	2 bytes
Last patch for this module	Variable
00000001 (Indicates last patch for this module)	4 bytes
.	.
.	.
.	.
Identifier for last module to be modified	2 bytes
Relative address of first patch for this module	2 bytes
Length (in bytes) of first patch for this module	2 bytes
First patch for this module	Variable
.	.
.	.
.	.
Relative address of last patch for this module	2 bytes
Length (in bytes) of last patch for this module	2 bytes
Last patch for this module	Variable
00000001 (Indicates last patch for this module)	4 bytes
ZZ (Indicates last module to be patched)	2 bytes

BLOCKING TABLE

The blocking table is constructed by Phase 7 only for PRFRM compilations. Phase 7 constructs a blocking table entry for each of the data control blocks that were opened by Phase 1. The blocking table contains the information required for deblocking compiler input and for blocking compiler output.

Each blocking table entry (24 bytes in length) has the following format:

Logical record length* (2 bytes)
Blocking factor (2 bytes)
Address of buffer 2 (4 bytes)
Address of buffer 1 (4 bytes)
Address of next logical record within the current buffer (4 bytes)
Address to or from which the next record is to be moved (4 bytes)
Number of logical records in current buffer that remain to be processed (2 bytes)
Indicates if a READ or WRITE has been issued for data set (1 byte)
Indicates whether data set has been previously referenced (1 byte)
*80 for SYSIN, SYSLIN, and SYSPUNCH; 121 for SYSPRINT.

BLDL TABLE

The BLDL table is constructed by Phase 7 only for PRFRM compilations. It is built using a BLDL macro-instruction. Phase 7 supplies, as a parameter of the BLDL macro-instruction, the address of a skeleton BLDL table. The skeleton build table contains: (1) the names (8 bytes per name) of the compiler components to which control may be transferred via an XCTL macro-instruction, and (2) a 28-byte field for each of the above names. The corresponding build routine completes the skeleton BLDL table by placing information into these 28-byte fields. This information is obtained from the data set directory of the partitioned data set containing the FORTRAN IV (E) compiler. This information (such as the physical location of each compiler component in the partitioned data set) is required for transferring control for PRFRM compilations from one component of the compiler to the next.

Each entry in the BLDL table is 36 bytes in length. The format of the BLDL table is as follows:

Compiler component (8 bytes)	Directory information for compiler component (28 bytes)
IEJFAAB0 (Phase 1-subsequent entries	Directory information for Phase 1 (subsequent entries)
IEJFAKA0 (Print buffer module)	Directory information for Print buffer module
IEJFEAA0 (Phase 7)	Directory information for Phase 7
IEJFGAA0 (Phase 10D)	Directory information for Phase 10D
IEJFJAA0 (Phase 10E)	Directory information for Phase 10E
IEJFJGA0 (Interlude 10E)	Directory information for Interlude 10E
IEJFLAA0 (Phase 12)	Directory information for Phase 12
IEJFNAA0 (Phase 14)	Directory information for Phase 14
IEJFNGA0 (Interlude 14)	Directory information for Interlude 14
IEJFPAA0 (Phase 15)	Directory information for Phase 15
IEJFPGA0 (Interlude 15)	Directory information for Interlude 15
IEJFRAA0 (Phase 20)	Directory information for Phase 20
IEJFVAA0 (Phase 25)	Directory information for Phase 25
IEJFXAA0 (Phase 30)	Directory information for Phase 30

APPENDIX D: INTERMEDIATE TEXT

Intermediate text is an internal representation of the source module from which the machine language instructions are produced. The conversion from intermediate text to machine language 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 dictionary and overflow table, and is referenced by the intermediate text. The dictionary and overflow table supplement the intermediate text in the generation of machine instructions by the various phases of the compiler.

Phases 10D and 10E create intermediate text for use as input to subsequent phases of the compiler. Intermediate text is created by Phase 10D for the following declarative statements:

- FORMAT
- SUBROUTINE or FUNCTION

Phase 10E creates intermediate text for all statement functions and executable statements in the source module and for FORMAT statements interspersed within the executable statements.

Phase 12 does not use the intermediate text during its processing; all of the remaining phases (14,15,20,25, and 30) do use the intermediate text during processing.

Phase 14 converts the FORMAT intermediate text to a form acceptable to IHCFCOME. It also inserts the addresses assigned by Phase 12 to variables, constants, etc., into the intermediate text. In addition, Phase 14 rearranges the intermediate text entries of READ/WRITE statements and inserts implied DO and end DO adjective codes into the intermediate text when an implied DO is encountered in a READ/WRITE statement.

Phase 15 reorders the sequence of intermediate text entries in statements that can contain arithmetic expressions, and modifies these entries to a format that closely resembles machine language instructions. Machine operation codes and registers (when required) are inserted in the intermediate text. Argument lists for external and function references are created by modifying the intermediate text for those statements. In-line function references are processed by generating the appropriate instruction format(s) and a word for the in-line function call.

Phase 20 modifies the intermediate text entries that represent subscript expressions. Registers are assigned to subscript expressions (once they have been initially computed) and are inserted in the text entries for those expressions.

Phase 25 uses the intermediate text in conjunction with the overflow table to generate the object module instructions.

Phase 30 uses the intermediate text to generate any error and warning messages and to process the END statement.

AN ENTRY IN THE INTERMEDIATE TEXT

The intermediate text is constructed by Phases 10D and 10E for some declarative statements, all statement functions, and all executable statements. Each statement is represented in the intermediate text by one or more intermediate text words. (An intermediate text word is four bytes long.) This word normally contains three fields (as illustrated in Figure 34).

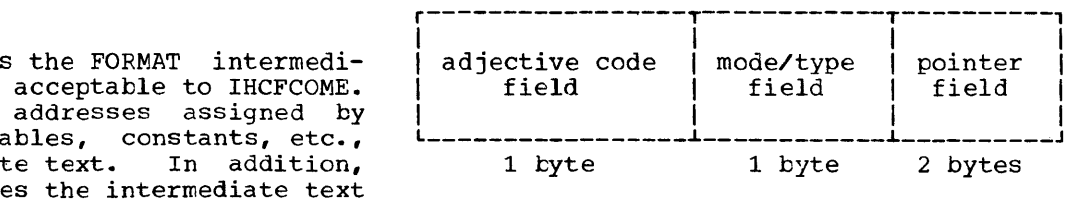


Figure 34. Intermediate Text Word Format

Adjective Code Field

The adjective code field in the initial intermediate text word indicates the type of statement for which the intermediate text entries are constructed, i.e.:

- Reserved word, e.g., DO,CALL, GOTO.
- Statement function (SF).
- Arithmetic.

The adjective codes in the subsequent intermediate text words for a statement indicate:

- Delimiters, i.e., + - * / ** ( ) ,
- The end of a statement (end mark)
- An error

The adjective code is composed of two hexadecimal digits. The various adjective codes possible (and their use) are indicated in Figure 35.

Mode/Type Field

The mode/type field indicates the mode and the type of a symbol; e.g., a real function for a function name, or dummy variable for the variable name. These mode/type codes are the same as those used in the dictionary entries (refer to Appendix C).

In the word with an end mark adjective code, another indicator may appear in the mode/type field. Normally, this field contains zeros; however, if any errors or warnings are detected in a statement, this field contains a hexadecimal 01.

If errors or warnings were detected, the error/warning message number appears in the mode/type field of the word inserted in the intermediate text to represent that error/warning. Errors and warnings are detected by Phases 10D, 10E, 12, 14, 15, and 20.

Pointer Field

The pointer field consists of the last two bytes of the intermediate text word. It normally contains a relative pointer to the dictionary or overflow table entry for the symbol with which the adjective code is associated, e.g., the term +A has a + adjective code and an associated pointer field that contains a relative pointer to the dictionary entry for A. The pointer field may also be used to contain either the increment of a DO or implied DO variable, or the internal statement number in the word containing the end mark or the error/warning adjective code.

The internal statement number is assigned during Phases 10D and 10E to each FORTRAN source statement. This number differs from the user-assigned statement number. It is assigned whether or not intermediate text is to be created for that statement; therefore, there may be gaps in the internal statement numbers appearing in the intermediate text. Errors in the source module may cause the same statement number to be assigned more than once. If the user has requested a source listing, the internal statement number assigned to each statement appears next to that statement in the listing.

An Example of an Intermediate Text Entry

For the statement

```
3   IF   (+19 - MART)   11, 7, 61
```

the intermediate text created by Phase 10E is:

adjective code field	mode/type field	pointer field
statement number	statement number	p(3)
arithmetic IF	00	0000
(	00	0000
unary +	integer constant	p(19)
-	integer variable	p(MART)
)	statement number	p(11)
'	statement number	p(7)
,	statement number	p(61)
end mark	00	internal statement number
1 byte	1 byte	2 bytes

p(x) indicates a pointer to the overflow table entry or the dictionary entry for x.

Unique Forms of Intermediate Text

When the intermediate text is created, there are four unique forms: the text for FORMAT statements, subscripted variables, COMMON statements, and EQUIVALENCE statements.

\L H\o i\w g\ h\	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0				.	(	)	=	' ARGU-MENT	[11]END MARK / N[10]	ILLEGAL	+	-	*	/	**	FUNC(	
1	AOP	UNARY MINUS		SAOP		SIZE OF ARRAY	END MARK						UNARY PLUS		[10]	' APOSTROPHE	
2			STM	IN-[10] LINE FUNC	ARITH-METIC IF	LM	$		BLANK								
3																	
4	S							BC[10]									
5	T			LCR						C		S U	M U				INTEGER
6	O									O M		B T	L T	D I			DOUBLE PRECISION
7	R								L	P		R	I	V			REAL
8	E			LCER					O A D	A R E	A D D	A C T	P L Y	I D E	SRDA[10]		
9					INTEGER	DOUBLE	REAL		COMMON	EQUIVA-LENCE	EXTER-NAL		DIMEN-SION			SUBROU-TINE	
A	FUNC-TICN	FORMAT	END DO	CON-TINUE	UNCONDI-TIONAL GO TC	COMPUT-ED GO TO	BACK-SPACE	REWIND	END FILE	WRITE BINARY	READ BINARY	WRITE BCD	READ BCD	DO	STMT. NO. DEF.		
B	END		CALL	SF		ARITH		BEGIN I/O LIST	END I/O LIST	RETURN	STOP	PAUSE	ARITH IF	IMP DO	ERROR MESS-AGE	WARNING MESS-AGE	
C																	
D																	
E																	
F																	

[10] Subject to change in later phases.
[11] The '08' end mark is a transient code that exists in Phases 10D and 10E only. It is used to generate the '16' end mark in intermediate text.

Figure 35. Adjective Codes as Used in Phases 10D and 10E

FORMAT STATEMENTS: For FORMAT statements, the adjective code field of the first intermediate text word of the statement indicates a FORMAT statement; the remaining two fields contain three bytes of the FORMAT statement card image. The remainder of the card image of the FORMAT statement appears in the following intermediate text words. For example, the statement;

```
12 FORMAT (F20.5,I6)
```

appears in the intermediate text as:

adjective code field	mode/type field	pointer field	
statement number	statement number	p(12)	
FORMAT	(	F	2
0	.	5	,
I	6	)	blank
blanks represent the remaining card columns to column 72 (each card column represents 1 byte)			
end mark	00	internal statement number	
1 byte	1 byte	2 bytes	

SUBSCRIPTED VARIABLE: When a subscripted variable is encountered in a source statement, an entry for a variable is made. That entry is followed by two additional intermediate text words to define the subscripted expression. The first word is of the form:

adjective code field	mode/type field	pointer field
SAOP	00	offset
1 byte	1 byte	2 bytes
SAOP represents the subscript arithmetic operator, and the offset represents a part of the array displacement. (Refer to Appendix E for a discussion of array displacement.)		

The second word is of the form:

adjective code field	mode/type field	pointer field
p(subscript information)		p(dimension information)
2 bytes		2 bytes
The first field contains a relative pointer to the subscript information in the overflow table if the subscripted expression contains variables. If the subscripted expression does not contain variables, this field contains zeros.		
The second field contains a relative pointer to the dimension information in the overflow table for the array that contains the subscripted expression. For example, if A (I,J) is an element in array A, the field contains the pointer to the dimension information for array A.		

The statement:

```
APPLE = A(POT,3) + B(2,1)
```

appears in the intermediate text as:

adjective code field	mode/type field	pointer field
arithmetic statement	mode/type of APPLE	p(APPLE)
=	mode/type of A	p(A)
SAOP	00	offset
p(subscript information)		p(dimension information)
+	mode/type of B	p(B)
SAOP	00	offset
00	00	p(dimension information)
end mark	00	internal statement number
1 byte	1 byte	2 bytes

COMMON STATEMENTS: An entry in COMMON intermediate text represents a variable or an array encountered in a COMMON source statement. Phase 12 references these entries (serially) and assigns addresses to them in the COMMON area. (The assignment of addresses is discussed in detail in the Phase 12 description.) Each entry has the form indicated below:

pointer to the variable or array entry in the dictionary	length of the name of the variable or array	not used
2 bytes	1 byte	1 byte

The first field contains the address of the dictionary entry for that variable or array.

The second field contains the length of the name of the variable or array in EBCDIC (Extended Binary Coded Decimal Interchange Code) characters. The length is used to determine in which chain of the dictionary the variable or array is to be entered.

Termination of all COMMON intermediate text is indicated by a two-byte termination indicator of the form:

0001
2 bytes

This termination indicator appears whether or not COMMON intermediate text exists.

An Example of COMMON Text: For the statement:

```
COMMON (A, R, ARNONN)
```

the COMMON intermediate text is:

p (A)	1	not used
p (R)	1	not used
p (ARNONN)	6	not used
2 bytes	1 byte	1 byte

EQUIVALENCE STATEMENTS: The EQUIVALENCE intermediate text is constructed by Phase 10D as a series of entries (one for each variable or array in an EQUIVALENCE group). Phase 12 references these entries (serially) and assigns addresses to them. (The assignment of addresses is discussed in detail in the Phase 12 description.)

Each entry in the EQUIVALENCE intermediate text has the following format:

pointer	size	offset or 0000
2 bytes	2 bytes	2 bytes

The first field is a pointer to the dictionary entry for the variable in question.

The second field contains the size of the variable in bytes, or the size of the array in bytes if the variable is dimensioned.

The third field contains the offset if this particular variable is subscripted, or 0000 if the variable is not subscripted.

Termination of an EQUIVALENCE group is indicated by a two-byte termination indicator of the following form:

0001
2 bytes

An Example of EQUIVALENCE Text: For the statement:

```
EQUIVALENCE (GRW,KEL),(RBJ(1,9),AMV(2,4))
```

there are two EQUIVALENCE groups:

- GRW,KEL
- RBJ (1,9),AMV(2,4)

where:

```
GRW is real
KEL is integer
RBJ is a real array dimensioned as (9,9)
AMV is a real array dimensioned as (9,4)
```

The EQUIVALENCE text is:

p(GRW)	4	0	Detail entry for GRW
p(KEL)	4	0	Detail entry for KEL
0001			EQUIVALENCE group termination indicator
p(RBJ)	324	288	Detail entry for RBJ
p(AMV)	144	112	Detail entry for AMV
0001			EQUIVALENCE group termination indicator
2 bytes	2 bytes	2 bytes	

MODIFYING INTERMEDIATE TEXT

The intermediate text is created by Phases 10D and 10E, and is modified by Phases 14, 15, and 20. This modification prepares the intermediate text for use by Phase 25 in the generation of machine language instructions. The modifications made to the intermediate text are discussed, phase by phase, in the following pages.

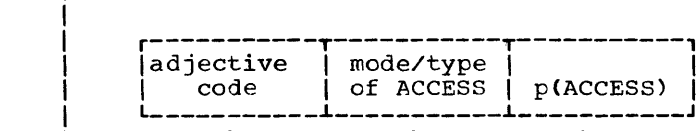
Phase 14

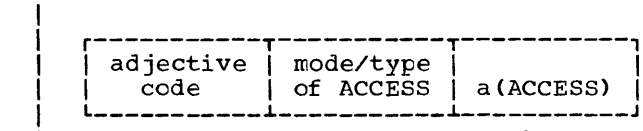
During Phase 14 processing, the intermediate text is modified in the following ways:

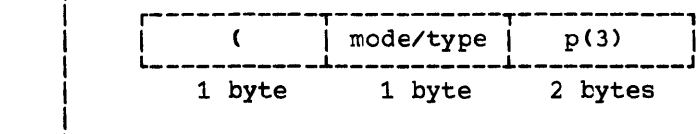
- Replacement of dictionary pointers.
- Modification of I/O statement intermediate text.
- Modification of computed GO TO intermediate text.
- Modification of RETURN intermediate text.

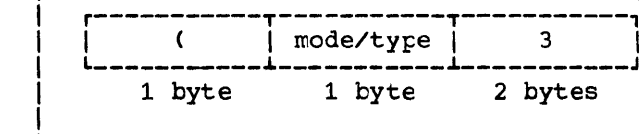
REPLACEMENT OF DICTIONARY POINTERS: Dictionary pointers in the intermediate text are replaced by information essential for the processing to be performed by subsequent phases of the compiler. The following examples illustrate this modification to intermediate text entries.

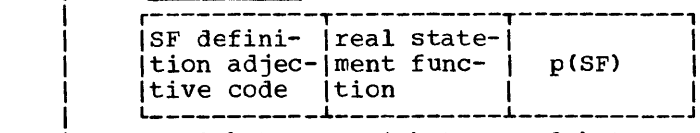
<table>
<tr><th>Input to Phase 14</th><th>Output from Phase 14</th></tr>
<tr><td>For:</td><td>the dictionary pointer is replaced by:</td></tr>
<tr><td>variables, constants, arrays, and external functions,

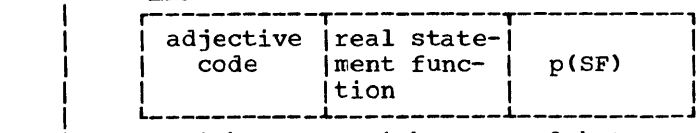
 
</td><td>the relative address assigned by Phase 12.

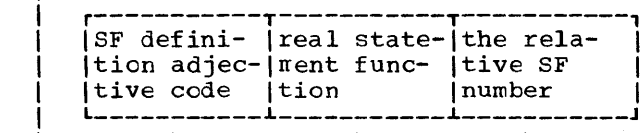
 
</td></tr>
<tr><td>data set reference numbers,

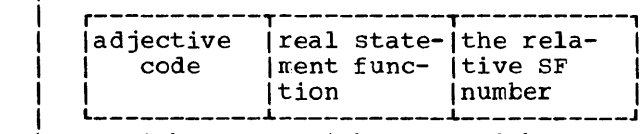
 
</td><td>the data set reference number.

 
</td></tr>
<tr><td>statement functions,

definition
 

use
 
</td><td>the SF number assigned by Phase 14.

 
</td></tr>
</table>

MODIFICATION OF I/O STATEMENT INTERMEDIATE TEXT: An I/O statement is modified in two ways. The begin I/O intermediate text word is inserted in the intermediate text for each element of an I/O list. An element is either an implied DO, or consecutive non-subscripted variables. Implied DOs are detected, and implied DO and end DO intermediate text words are entered in the text. An end I/O is placed at the end of the I/O list.

These modifications are illustrated in Figures 36 and 37, which show an indexed I/O list for a 2-dimensional array as it appears as input to and output from Phase 14. The intermediate text in these figures is developed from the following I/O statement:

```
WRITE (N) ((A(I,J),J=1,10),I=1,15)
```

WRITE	00	0000
(	integer variable	p(N)
)	00	0000
(	00	0000
(	real subscripted variable	p(A)
SAOP	00	Offset
p(subscript)		p(dimension)
,	integer variable	p(J)
=	immediate DO parameter	1
,	immediate DO parameter	10
,	parameter	1
)	00	0000
,	integer variable	p(I)
=	immediate DO parameter	1
,	immediate DO parameter	15
,	immediate DO parameter	1
)	00	0000
end mark	00	internal statement number

Figure 36. Example of Input to Phase 14

WRITE	00	0000
(	integer variable	address(N)
end mark [1]	00	0000
implied DO	00	0000
,	integer variable	address(I)
=	immediate DO parameter	1
,	immediate DO parameter	15
,	immediate DO parameter	1
implied DO	00	0000
,	integer variable	address(J)
=	immediate DO parameter	1
,	immediate DO parameter	10
,	immediate DO parameter	1
begin I/O	00	0000
SAOP	00	Offset
p(subscript)		p(dimension)
(	real subscripted variable	address(A)
end DO	00	0000
end DO	00	0000
end I/O	00	0000
end mark	00	internal statement number

[1] An end mark is inserted prior to the I/O list. This allows Phase 20 to treat the I/O list as a separate statement.

Figure 37. Example of Output from Phase 14

MODIFICATION OF COMPUTED GO TO STATEMENTS: During the Phase 14 processing, a count of the number of statement numbers in the computed GO TO statement is inserted into the intermediate text for that statement. This simplifies the processing of this intermediate text for the following phases. The intermediate text is rearranged so that the word containing the integer variable precedes the count word.

A computed GO TO statement such as:

```
GO TO (11,11,42,23,99),I
```

appears in the input to Phase 14 as:

adjective code field	mode/type field	pointer field
computed GO TO	00	0000
(	statement number	p(11)
,	statement number	p(11)
,	statement number	p(42)
,	statement number	p(23)
,	statement number	p(99)
)	00	0000
,	integer variable	p(I)
end mark	00	internal statement number

The output of Phase 14 for the above illustrated computed GO TO is:

adjective code field	mode/type field	pointer field
computed GO TO	00	0000
,	integer variable	a(I)
count	00	5
(	statement number	p(11)
,	statement number	p(11)
,	statement number	p(42)
,	statement number	p(23)
,	statement number	p(99)
)	00	0000
end mark	00	internal statement number

MODIFICATION OF RETURN STATEMENT INTERMEDIATE TEXT: If a RETURN statement appears within a main program, Phase 14 modifies the adjective code field so that a STOP is indicated. If the RETURN statement is not within the main program, no modification is made.

Phase 15

During Phase 15 processing, the following intermediate text modifications are made:

- Replacement of adjective codes and mode/type codes.
- Reordering of intermediate text.

REPLACEMENT OF ADJECTIVE CODES AND MODE/TYPE CODES: During the processing of arithmetic expressions, Phase 15 replaces the adjective codes (within the intermediate text entries for arithmetic expressions) by actual machine operation codes. Phase 15 also assigns registers to the operands in arithmetic expressions (when required); the corresponding register numbers are inserted in the mode/type code fields of the intermediate text that represents those expressions.

The result of the above modification is a transformation of the intermediate text entries for arithmetic expressions into a form that closely resembles the RX instruction format.

The following example indicates the replacement of adjective codes by machine operation codes, and the replacement of mode/type codes by registers. The simple arithmetic statement

```
PRI = +VATE - VAR
```

appears in the input to Phase 15 as:

adjective code field	mode/type code field	pointer field
arithmetic statement	real variable	a(PRI)
=	00	0000
unary plus	real variable	a(VATE)
-	real variable	a(VAR)
end mark	00	internal statement number
1 byte	1 byte	2 bytes

The pointer field contains the address of the resultant field of the arithmetic statement.

The output from Phase 15 for this statement is:

adjective code field	mode/type code field		pointer field
arithmetic statement	real variable		a(PRI)
L	reg.#3	variable	a(VATE)
S	reg.#3	variable	a(VAR)
ST	reg.#3	variable	a(PRI)
end mark	00		internal statement number
1 byte	1 byte		2 bytes

The first operand VATE, is loaded into register #3. The second operand, VAR, is subtracted from VATE. The result is stored in the resultant field, PRI.

In addition, registers are assigned and are inserted in the mode/type code field of the following:

- Intermediate text entries for exponentiation.
- Intermediate text entries for in-line functions, referenced subprograms, and statement function calls.
- Intermediate text entries for subscript expressions.

The following examples illustrate this modification to the intermediate text.

Input To Phase 15	Output From Phase 15
For:	Phase 15 assigns:
exponentiation, |** | mode/type information | a(POWER) | 1 byte, 1 byte, 2 bytes	a register to contain the result of the required library subprogram execution. |** | 0 | result reg | a(POWER) | 1 byte, 1 byte, 2 bytes
in-line functions, |in-line function adj. code | not used | code number of in-line funct. | |F(| not used | a(argument) | 1 byte, 1 byte, 2 bytes	one or two registers (depending on the specific in-line function) to be used as argument registers. The register specified in the R1 field is used as the result register. |Load | R1 | not used | a(argument) | |in-line function adj. code | R2 | R1 | code number of in-line funct. | 1 byte, 1 byte, 2 bytes
subscript expressions, |subscript adj. code | mode/type information | Offset | 1 byte, 1 byte, 2 bytes	a work register (to be used by Phase 20) to aid in the computation of the subscript expression. |subscript adj. code | 0 | work reg. | Offset | 1 byte, 1 byte, 2 bytes

REORDERING OF INTERMEDIATE TEXT: Phase 15 reorders the intermediate text entries within arithmetic expressions so that the object module instructions produced by subsequent phases are generated according to a hierarchy of operators.

The following example indicates this reordering process.

The statement:

```
DGM = BCR*(WRG+WAR)
```

appears in the input to Phase 15 as:

adjective code	mode/type code	pointer field
arithmetic	real variable	a(DGM)
=	real variable	a(BCR)
*	00	0000
(	real variable	a(WRG)
+	real variable	a(WAR)
)	00	0000
end mark	00	internal statement number
1 byte	1 byte	2 bytes

The output from Phase 15 for this statement is:

adjective code	mode/type code		pointer field
arithmetic	real variable		a(DGM)
LE	register 6	variable information	a(WRG)
AE	register 6	variable information	a(WAR)
ME	register 6	variable information	a(BCR)
STE	register 6	variable information	a(DGM)
end mark	00		internal statement number
1 byte	1 byte		2 bytes

Phase 20

Phase 20 optimizes the intermediate text entries for subscript expressions. This optimization consists of modifying portions of existing subscript intermediate text and creating new subscript intermediate text for literals that are generated during the subscript optimization process. The changes made to subscript intermediate text will be discussed by examining a general subscript expression as it appears in the input to Phase 20 and by examining the subscript intermediate text output from Phase 20 for this expression.

SUBSCRIPT INTERMEDIATE TEXT INPUT: The intermediate text input to Phase 20 for a general expression is shown in Figure 38.

SUBSCRIPT INTERMEDIATE TEXT OUTPUT: Subscript intermediate text output from Phase 20 depends on the previous optimization (if any) of the subscript expression. Three adjective codes are used to indicate the different conditions that can be present in subscript intermediate text output. These conditions are explained in the following paragraphs.

adjective code field	mode/type field		pointer field
adjective code	O	W	offset
p(subscript)			p(dimension)
OP	R	Type	a(variable)
1 byte	1 byte		2 bytes

Adjective code contains the adjective code for a subscripted variable portion of text.

O contains a zero value.

W contains a work register assigned by Phase 15.

Offset contains the value of the offset portion of the array displacement.

p(subscript) contains the pointer to subscript information in the overflow table for this expression.

p(dimension) contains the pointer to dimension information in the overflow table for this expression.

OP contains the operation code assigned by Phase 15.

R contains a register assigned by Phase 15.

Type contains the residual (since it is no longer necessary) type information for the subscripted variable.

a(variable) contains the address of the subscripted variable.

Figure 38. Subscript Intermediate Text Input Format

SAOP (Subscript Arithmetic Operator) Adjective Code: This code indicates that a subscript expression has not been previously optimized, and that an offset literal was not generated for the value resulting from the addition of the offset portion of the array displacement to the subscripted variable address displacement. Subscript text output associated with an SAOP adjective code is shown in Figure 39.

adjective code field	mode/type field		pointer field
SAOP	N	W	offset
p(subscript)			a(C1*L)
a(C2*D1*L)			a(C3*D1*D2*L)
OP	R	X	a(variable)
1 byte	1 byte		2 bytes

SAOP contains an adjective code designating the form of the intermediate subscript text.

N contains the number of dimensions of the subscripted variable.

a(C1*L), a(C2*D1*L), and a(C3*D1*D2*L) contain the addresses of the literals that combine to form the CDL portion (see Appendix E) of the array displacement. N determines which addresses must appear. For example, if N is 1, only a(C1*L) appears. (If the first literal, C1*L, is a power of 2, that power appears instead of the address of that literal.)

X contains the register assigned to the subscript expression for computation by Phase 20.

Note: All other entries are as defined in Figure 38.

Figure 39. Subscript Intermediate Text Output From Phase 20 -- SAOP Adjective Code

XOP (Offset Literal) Adjective Code: This code indicates that the subscript expression has not been previously assigned a register and that an offset literal was generated for the value resulting from the addition of the offset portion of the array displacement to the displacement of the subscripted variable address. The subscript intermediate text output associated with an XOP adjective code is shown in Figure 40.

AOP (Arithmetic Operator Without Subscript) Adjective Code: This code indicates that the subscript expression has previously been assigned a register. The subscript intermediate text output associated with an AOP adjective code is shown in Figure 41.

adjective code field	mode/type field		pointer field
XOP	N	W	a(generated literal)
p(subscript)			a(C1*L)
a(C2*D1*L)			a(C3*D1*D2*L)
OP	R	X	a(variable)
1 byte	1 byte		2 bytes

XOP contains an adjective code designating the form of the subscript intermediate text.

a(generated literal) contains the address of the offset literal generated by Phase 20.

Note: All other entries are as defined in Figures 38 and 39.

Figure 40. Subscript Intermediate Text Output from Phase 20 -- XOP Adjective Code

adjective code field	mode/type field		pointer field
AOP	O	B	offset
OP	R	X	a(variable)
1 byte	1 byte		2 bytes

AOP contains an adjective code designating the form of subscript intermediate text.

O contains a zero value.

B contains an indicator. A hexadecimal 0 indicates that the actual offset is in the offset field. A hexadecimal F indicates that the address of the generated offset literal appears in the offset field.

Note: All other entries are as in Figures 38 and 39.

Figure 41. Subscript Intermediate Text Output from Phase 20 -- AOP Adjective Code

APPENDIX E: ARRAY DISPLACEMENT COMPUTATION

Array displacement is the distance between the first element in an array and a specified element to be referenced from the array. To increase compilation efficiency, the array displacement is divided into portions and computed during different phases. To tie these separate computations into one coordinated presentation, the method of array displacement computation is developed in the following text.

Before discussing the actual computation, it is desirable to understand how an element is referenced in a 1-, 2-, and 3-dimensional array.

ONE DIMENSION

Assume a 1-dimensional array of five elements, expressed as A(5). To reference any given element in this array, the only factor to be considered is the length of each element. The third element, for example, is two element lengths from the beginning of the array.

TWO DIMENSIONS

For a 2-dimensional array, A(3,2), an element can no longer be thought of as a single array element. Instead, each element in a 2-dimensional array consists of the number of array elements designated by the first number in the subscript expression used to dimension the array. For reference, an element in a 2-dimensional array will be called a dimension part. For example, in the array of A(3,2):

```
A(1,1)  A(2,1)  A(3,1)-┐ - Dimension Part
┌----------------------┘
└>A(1,2)  A(2,2)  A(3,2)   - Dimension Part
```

the first dimension part consists of A(1,1), A(2,1), and A(3,1). Note that the number of elements in each dimension part is the same as the first number (3) in the subscript expression used to dimension array A.

Dimension parts are consistent in length. Length is determined by multiplying the number of elements in a dimension part by the array element length. The resulting value is considered a dimension factor for the following discussion. (If the element length in array A is 4, the dimension factor is 3 times 4, or 12.) The dimension factor plays a significant role in referencing a specific element in a 2-dimensional array.

Before discussing how a specified element is referenced, the hexadecimal number scheme used to address an array element must be considered. The first digit of the hexadecimal number scheme (as used in the compiler) is zero. The 16 hexadecimal digits are:

0,1,2,3,4,5,6,7,8,9,A,B,C,D,E, and F.

Consider that the element A(1,2) is to be referenced from the array dimensioned as A(3,2). Observation shows one dimension part must be bypassed in order to reference the specified element. The computation to reference this element requires the values in the subscript expression (1,2). Each number must be decremented by 1 to compensate for the zero-addressing scheme used by the compiler. This leaves an expression of (0,1). The second number (1) dictates the number of dimension parts to be bypassed in order to arrive at the dimension part in which the specified element is located. Once this dimension part is found, the first number (0) indicates the number of elements in that dimension part that must be bypassed to reference the specified element.

THREE DIMENSIONS

The same reasoning can be projected into a 3-dimensional array. For a 3-dimensional array, A(3,2,3), an element can neither be considered a single array element, nor thought of as a dimension part. Each element in a 3-dimensional array consists of the number of dimension parts designated by the second number in the subscript expression used to dimension the array. For reference, therefore, an element in a 3-dimensional array will be called a dimension section. For example, in the array A(3,2,3):

```
 Dimension Section
  A(1,1,1)   A(2,1,1)   A(3,1,1)┐ - Dim.Part
                                │
┌───────────────────────────────┘
└>A(1,2,1)   A(2,2,1)   A(3,2,1)┐ - Dim.Part
                                │
┌───────────────────────────────┘
│
│Dimension Section
└>A(1,1,2)   A(2,1,2)   A(3,1,2)┐ - Dim.Part
                                │
┌───────────────────────────────┘
└>A(1,2,2)   A(2,2,2)   A(3,2,2)┐ - Dim.Part
                                │
┌───────────────────────────────┘
│
│Dimension Section
└>A(1,1,3)   A(2,1,3)   A(3,1,3)┐ - Dim.Part
                                │
┌───────────────────────────────┘
└>A(1,2,3)   A(2,2,3)   A(3,2,3)  - Dim.Part
```

the first dimension section consists of the dimension part beginning with A (1,1,1) and the dimension part beginning with A(1,2,1). In this example, we have three dimension sections, as specified by the third number in the subscript expression used to dimension the array.

Again, the length of the dimension sections is consistent. The length, in this case, is determined by multiplying the number of elements in a dimension part by the number of dimension parts by the array element length. The resulting value is considered a dimension multiplier for the following discussion. (If the element length in array A is 4, the dimension multiplier is 3 times 2 times 4 or 24.)

Consider that the element A (2,2,3) is to be referenced from the array dimensioned as A (3,2,3). Observation shows two dimension sections, one dimension part, and one array element must be bypassed in order to obtain the specified element. The computation to reference this element requires the values in the subscript expression (2,2,3). Each number must be decremented by 1 to compensate for the zero-addressing scheme used by the compiler. This leaves an expression of (1,1,2). The third number (2) indicates the number of dimension sections to bypass in order to arrive at the dimension section in which the specified element is located. The second number (1) indicates the number of dimension parts, within the referenced dimension section, that must be bypassed to arrive at the dimension part in which the specified element is located. Once this dimension part is found, the first number (1) indicates the number of elements in that dimension part that must be bypassed to reference the specified element. The preceding example is illustrated in Figure 42.

This concept of how a specified element is referenced from an array is generalized in the following text.

<u>General Subscript Form</u>

The general subscript form (C1*V1+J1,C2*V2+J2,C3*V3+J3) refers to some array, A, with dimensions (D1, D2, D3). The required number of elements is specified by (C1*V1+J1); (C2*V2+J2) *D1; and (C3*V3+J3) *D1*D2, representing the first, second, and third subscript parameters multiplied by the pertinent dimension information for each parameter. Therefore, the required number of elements for the general subscript form is:

```
(C1*V1+J1)+(C2*V2+J2)*D1+(C3*V3+J3)*D1*D2
```

```
┌──────────────────────────────────────────────────────────────────────────┐
│             A(2,2,3)                                                     │
│                 │                                                        │
│                 │ Zero-addressing adjustment                             │
│                 V                                                        │
│             A(1,1,2)                                                     │
│               │ │ │                                                      │
│               │ │ │                                                      │
│               │ │ └───> 2 dimension sections ┐                           │
│               │ │                            │ Must be bypassed to       │
│               │ └─────> 1 dimension part     ┤                           │
│               │                              │ obtain specified element  │
│               └───────> 1 array element      ┘                           │
└──────────────────────────────────────────────────────────────────────────┘
```
Figure 42. Referencing a Specified Element in Array

Array Displacement

The array displacement for a subscript expression, specifically stated, is the required number of array elements multiplied by the array element length. Therefore, the array displacement is:

```
[(C1*V1+J1)+(C2*V2+J2)*D1+

  (C3*V3+J3)*D1*D2)]*L
```

Because of the zero-addressing scheme, the displacement is:

```
(C1*V1+J1-1)*L+(C2*V2+J2-1)*D1*L+
(C3*V3+J3-1)*D1*D2*L
```

This expression can be rearranged as:

```
(C1*V1*L+C2*V2*D1*L+C3*V3*D1*D2*L)+
[(J1-1)*L+(J2-1)*D1*L+(J3-1)*D1*D2*L)]
```

The first portion of the array displacement is referred to as the CDL (constant, dimension, length) portion and is derived from:

```
C1*V1*L+C2*V2+D1*L+C3*V3*D1*D2*L
```

V1, V2, and V3 are the variables of the expression and cannot be computed until the execution of the object module. This leaves the following components, which constitute the CDL portion of the displacement:

C1*L is the first component,
C2*D1*L is the second component, and
C3*D1*D2*L is the third component.

The second portion of the array displacement:

```
(J1-1)*L+(J2-1)*D1*L+(J3-1)*D1*D2*L
```

is known as the offset portion and is calculated by Phase 10E. The offset is calculated using the following formulas for 1-, 2-, and 3- dimensional arrays.

```
OFFSET=[J1-1]*Length
                                   1-dimensional

OFFSET=[(J1-1)+(J2-1)*D1]
                    *Length        2-dimensional

OFFSET=[(J1-1)+(J2-1)*D1
     +(J3-1)*D1*D2]*Length         3-dimensional
```

This calculation is performed and the result is entered in the offset field of the intermediate text entry for that subscript. Refer to Appendix D for the intermediate text format.

The CDL components are calculated during Phase 20. If the CDL component is a power of 2, that power replaces the offset field in the intermediate text entry. If the CDL component is not a power of 2, a literal is formed and assigned an address (by Phase 20). The address of the literal is then entered in the offset field of the intermediate text entry. Refer to Appendix D for the intermediate text form and content.

Phase 25 combines the CDL components, the variables, and the offset to produce the array displacement. The procedure is as follows: the first component of the CDL multiplied by the first variable of the subscript expression (C1*L)*V1; plus the second component of the CDL multiplied by the second variable of the subscript expression (C2*D1*L)*V2, plus the third component of the CDL multiplied by the third variable of the subscript expression (C3*D1*D2*L)*V3; plus the offset:

```
(J1-1)*L+(J2-1)*D1*L+(J3-1)*D1*D2*L.
```

The following tables are used by the object module to execute the instructions generated by the compiler:

- Branch list table for referenced statement numbers
- Branch list table for SF expansions and DO statements
- Argument list table for subprogram and SF calls
- Base value table

The following discussions describe the use and format of each table.

BRANCH LIST TABLE FOR REFERENCED STATEMENT NUMBERS

Phase 12 allocates storage for the branch list table for referenced statement numbers and assigns a relative position (relative to the start of the branch table) to each executable statement that is referenced by other statements. Phase 25 inserts the relative addresses, for these statements, into the positions dictated by Phase 12. The table is used, at object time, by the instructions generated to branch to executable statements.

Each entry in the table is the address of a referenced statement number. The format of the branch list table for referenced statement numbers is illustrated in Figure 43.

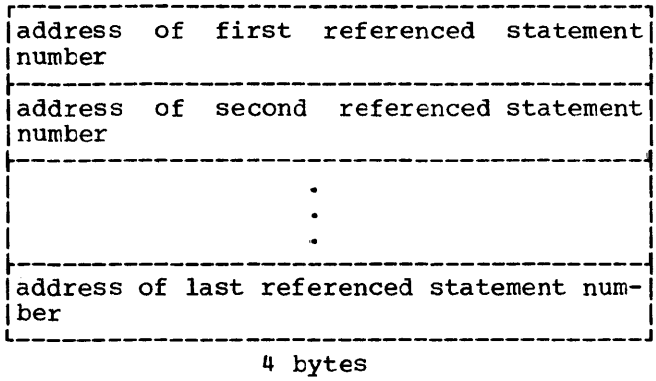


Figure 43. Format of Branch List Table for Referenced Statement Numbers

BRANCH LIST TABLE FOR SF EXPANSIONS AND DO STATEMENTS

Phase 20 allocates storage for the branch list table for SF (statement function) expansions and DO statements. During Phase 25 processing, the relative addresses for the first executable instructions in the SF expansions and DO loops are inserted into locations relative to the start of the branch table. The locations for the SF expansions were determined by Phase 14; the locations for the DO loops are determined by Phase 25. The table is used, at object time, either by the instructions generated to reference SF expansions or by the instructions generated to control the iteration of DO loops.

Each entry in the table is either the address of the first instruction in an SF expansion or the address of the second instruction in a DO loop. (The first instruction of the DO loop initializes the DO counter.) The format and organization of the branch list table for SF expansions and DO statements is illustrated in Figure 44.

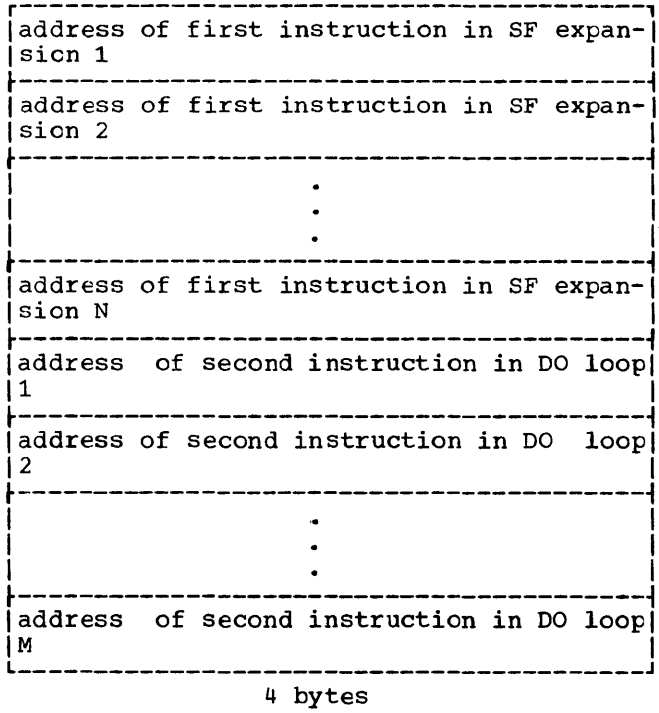


Figure 44. Format of Branch List Table for SF Expansions and DO Loops

All SF definitions must appear prior to the executable statements (this includes DO statements) in a source module. Therefore, Phase 25 encounters all the SF adjective codes prior to the first DO statement adjective code. This accounts for the placement of all SF expansion addresses into the branch table before the first DO loop address.

ARGUMENT LIST TABLE FOR SUBPROGRAM AND SF CALLS

Phase 20 allocates storage for the argument list table for the arguments of subprogram and SF calls. During Phase 20 processing, the relative addresses of the above arguments are inserted into the argument list table. The starting address of the first argument of each argument list is passed as part of the intermediate text to Phase 25 (the total number of SFs is passed in the communication area).

Each entry in the argument list table is either the address of an argument used in a subprogram or the address of an argument used in an SF. Entries are made in the table as Phase 20 encounters each subprogram or SF reference. The format and organization of the argument list table is illustrated in Figure 45.

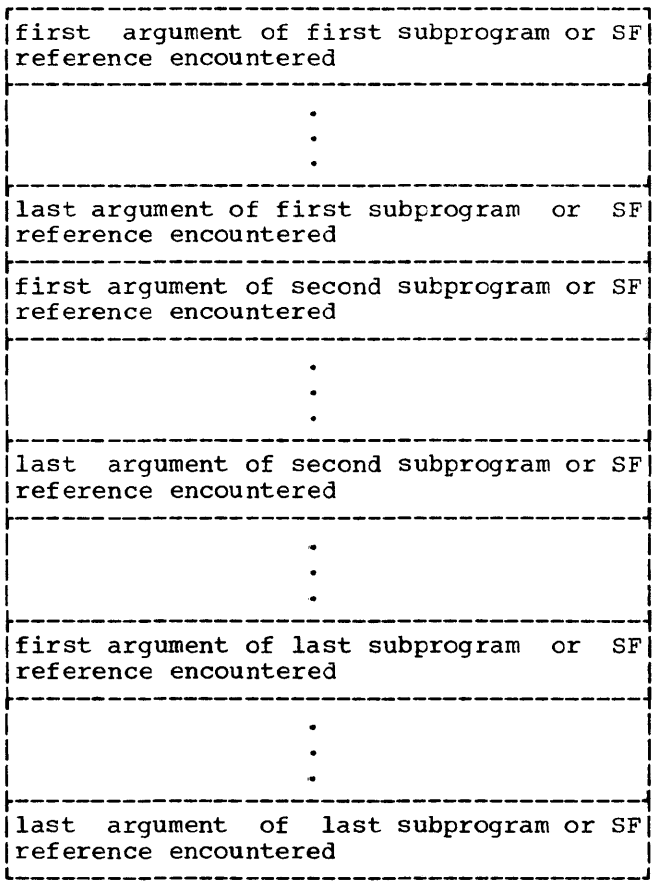


Figure 45. Format of Argument List Table for Subprogram and SF Calls

BASE VALUE TABLE

The base value table is generated by the various phases of the compiler as base registers are required by the object coding. The table is assembled in its final form by Phase 25. The compiler-generated instructions that load base registers, at object time, use the base value table in order to obtain the proper base register values.

Figure 46 illustrates the format and organization of the base value table.

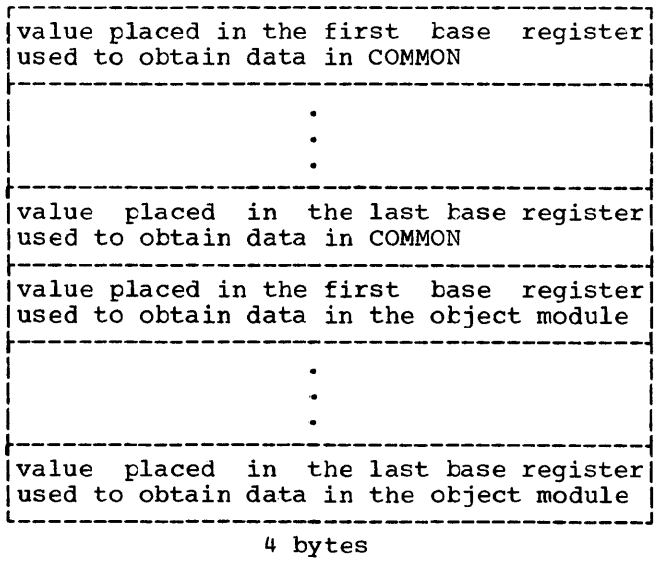


Figure 46. Format of Base Value Table

The object module, compiled from the FORTRAN source module, must be first processed by the linkage editor prior to execution on the IBM System/360. The linkage editor must combine certain FORTRAN library subprograms with the object module to form an executable load module. They are:

- IHCFCOME (Object-time I/O source statement processor).
- IHCFIOSH (Object-time FORTRAN I/O data management interface).
- IHCIBERR (Object-time source statement error processor).

IHCFCOME

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

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

In addition, IHCFCOME processes object-time errors detected by 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, IHCFCOME, IHCFIOSH, IHCIBERR, and any required subprograms.)

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

The routines of IHCFCOME are divided into the following categories:

- READ/WRITE routines.
- I/O device manipulation routines.
- Write-to-operator routines.
- Utility routines.

Charts 13, 14, and 15 illustrate the overall logic and the relationship among the routines of IHCFCOME. Table 26, the routine directory, lists the routines and their functions.

Note: IHCFCOME itself does not perform the actual reading from or writing onto data sets, or I/O device manipulation. It submits requests for such operations to an I/O interface module IHCFIOSH (that is, FIOCS#) by means of an implied external reference. IHCFIOSH, in turn, interprets the requests and submits them to the appropriate BSAM (basic sequential access method) routines for execution.

READ/WRITE Routines

For the implementation of READ and WRITE statements, IHCFCOME consists of the following three sections:

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

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

- As a READ/WRITE operation requiring a format.
- As a READ/WRITE operation not requiring a format.

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

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

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

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

The action taken by IHCFCOME when the various format codes are encountered is illustrated in Table 21.

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

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

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

READ/WRITE Requiring a Format: In processing a list item for a READ requiring a format, the I/O list section passes control to the conversion routine associated with the conversion code for the list item. (The appropriate conversion routine is determined by the portion of IHCFCOME that scans the FORMAT statement associated with the READ statement. The selection of the conversion routine depends on the conversion code of the list item being processed.) The conversion routine obtains data from an input buffer and converts the data to the form dictated by the conversion code. The converted data is then moved into the main storage address assigned to the list item.

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

Table 21. IHCFCOME FORMAT Code Processing

FORMAT Code	Description	Type	Corresponding Action Upon Code by IHCFCOME
	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	conversions	conversion	Exit to the load module 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 the current position in the I/O buffer, the scale factor, and the values of w and d. Upon return from the conversion routine the current field count is tested. If it is greater than 1, another exit is made to the load module to obtain the address of the next list item.
)	group end	control	Test group count. If greater than 1, repeat format codes in group; otherwise continue to process FORMAT statement from current position.
/	record end	control	Input or output one record via IHCFIOSH and READ/WRITE macro-instruction.
	end of statement	control	If no I/O list items remain to be transmitted, return control to the load module to link to the closing section; if list items remain, input or output one record using IHCFIOSH and READ/WRITE macro-instruction. Repeat format codes from last left parenthesis.

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

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

If the conversion code is to be repeated and if an array is being processed, the I/O list section computes the main storage address of the next element in the array. The conversion routine that processed the previous element is then given control. This procedure is repeated until either all the array elements associated with a specific conversion code are processed or the input data for the READ statement is exhausted.

If the conversion code is not to be repeated, control is passed to the scan portion of IHCFCOME to continue the scan of the FORMAT statement. If the scan portion determines that a group of conversion codes is to be repeated, the conversion routines corresponding to those codes are applied to the next portion of the input data. This procedure is repeated until either the group count is exhausted or the input data for the READ statement is exhausted.

If a group of conversion codes is not to be repeated and if the end of the FORMAT statement is not encountered, the next format code is obtained. For a control type code, control is passed to the associated control routine to perform the indicated operation. For a conversion type code, control is returned to the load module if the previous list item was a variable. The load module again branches to the I/O list section and passes, as an argument, the main storage address assigned to the next list item. Control is then passed to the conversion routine associated with the new conversion code. The conversion routine then processes the data for this list item. If the data that was just converted was placed into an element of an array and if the entire array has not been filled, the I/O list section computes the main storage address of the next element in the array and passes control to the conversion routine associated with the new conversion code. The conversion routine then processes the data for this array element. Subsequent I/O list processing for a READ requiring a format proceeds at the point where the field count is examined.

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

If the operation is a WRITE requiring a format, the I/O list section processing is similar to that for a READ requiring a format. The main difference is that the conversion routines obtain data from the main storage addresses assigned to the list items rather than from an input buffer. The converted data is then transferred to an output buffer. If all the list items have not been converted and transferred prior to the encounter of the end-of-the FORMAT statement, control is passed to IHCFIOSH. IHCFIOSH writes (via the WRITE macro-instruction) the contents of the current output buffer onto the output data set. The conversion codes starting from the last left parenthesis are then repeated for the remaining list items.

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

If the operation is a WRITE not requiring a format, the I/O list section processing is similar to that described for a READ not requiring a format. The main difference is that the data is obtained from the main storage addresses assigned to the list items and is then moved to an output buffer.

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

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

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

If the operation is a WRITE not requiring a format, the closing section inserts: (1) the record count (i.e., the number of records in the logical record) into the control word of the I/O buffer to be written, and (2) an end-of-logical-record indicator into the last record of the I/O buffer being written. The closing section then branches to IHCFIOSH. IHCFIOSH writes (via the WRITE macro-instruction) the contents of this I/O buffer onto the output data set. IHCFIOSH then returns control to the closing section. The closing section, in turn, returns control to the load module to continue load module exectuion.

Examples of IHCFCOME READ/WRITE Statement Processing

The following examples illustrate the opening section, I/O list section, and closing section processing performed by IHCFCOME for the operations of READ and WRITE, requiring or not requiring a format.

READ REQUIRING A FORMAT: The processing performed by IHCFCOME for the following READ statement and FORMAT statement is illustrated in Table 22.

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

Table 22. IHCFCOME Processing for a READ Requiring a Format

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

WRITE REQUIRING A FORMAT: The processing performed by IHCFCOME for the following WRITE statement and FORMAT statement is illustrated in Table 23.

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

Table 23. IHCFCOME Processing for a WRITE Requiring a Format

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

READ NOT REQUIRING A FORMAT: The processing performed by IHCFCOME for the following READ statement is illustrated in Table 24.

```
READ (5) X,Y,Z
```

Table 24. IHCFCOME Processing for a READ Not Requiring a Format

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

WRITE NOT REQUIRING A FORMAT: The processing performed by IHCFCOME for the following WRITE statement is illustrated in Table 25.

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

Table 25. IHCFCOME Processing for a WRITE Not Requiring a Format

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

I/O Device Manipulation Routines

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

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

The BACKSPACE statement is processed in a similar fashion. However, before control is returned to the calling routine, it is determined whether the record backspaced over is an element of a data set that does not require a format. If the record is an element of such a data set, that record is read into an I/O buffer and the record count is obtained from its control word. Backspace control requests, equal in number to the record count, are then issued and control is returned to the calling routine. If the record is not an element of such a data set, control is returned directly to the calling routine.

Write-to-Operator Routines

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

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

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 IHCFCOME perform the following functions:

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

PROCESSING OF ERROR MESSAGES: Error message processing routine (IBFERR) 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 also of writing the message. If the type of error requires load module termination, control is passed to the termination routine of IHCFCOME; if not, control is returned to the calling routine.

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

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

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

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

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

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

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

- Calling IHCFIOSH to check (via the CHECK macro-instruction) outstanding write requests.
- Issuing a SPIE macro-instruction with no parameters indicating that the FORTRAN object module no longer desires to give special treatment to program interruptions and does not want maskable interruptions to occur.
- Returning to the operating system supervisor.

Chart 13. IHCFCOME Overall Logic Diagram and Utility Routines

UTILITY ROUTINES

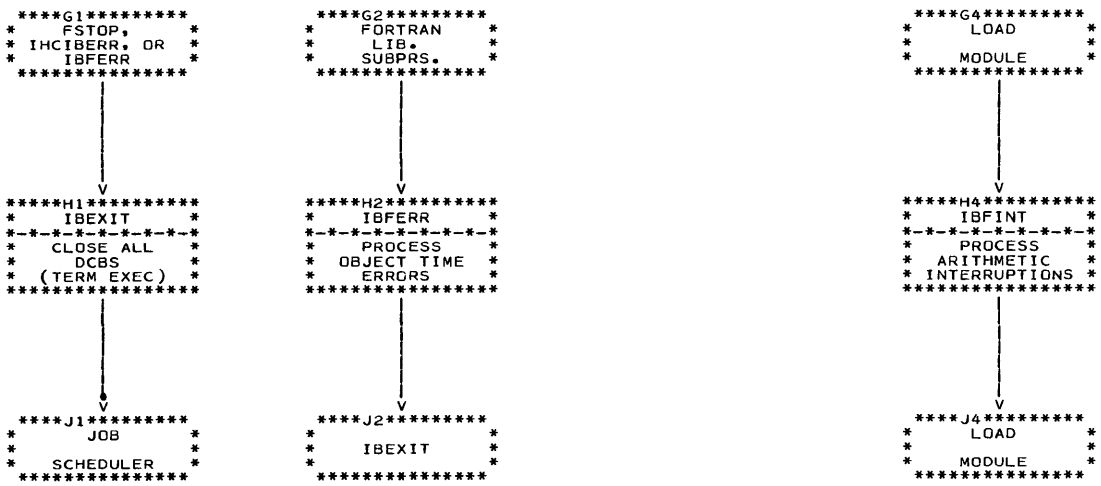


Chart 14. Implementation of READ/WRITE Source Statements

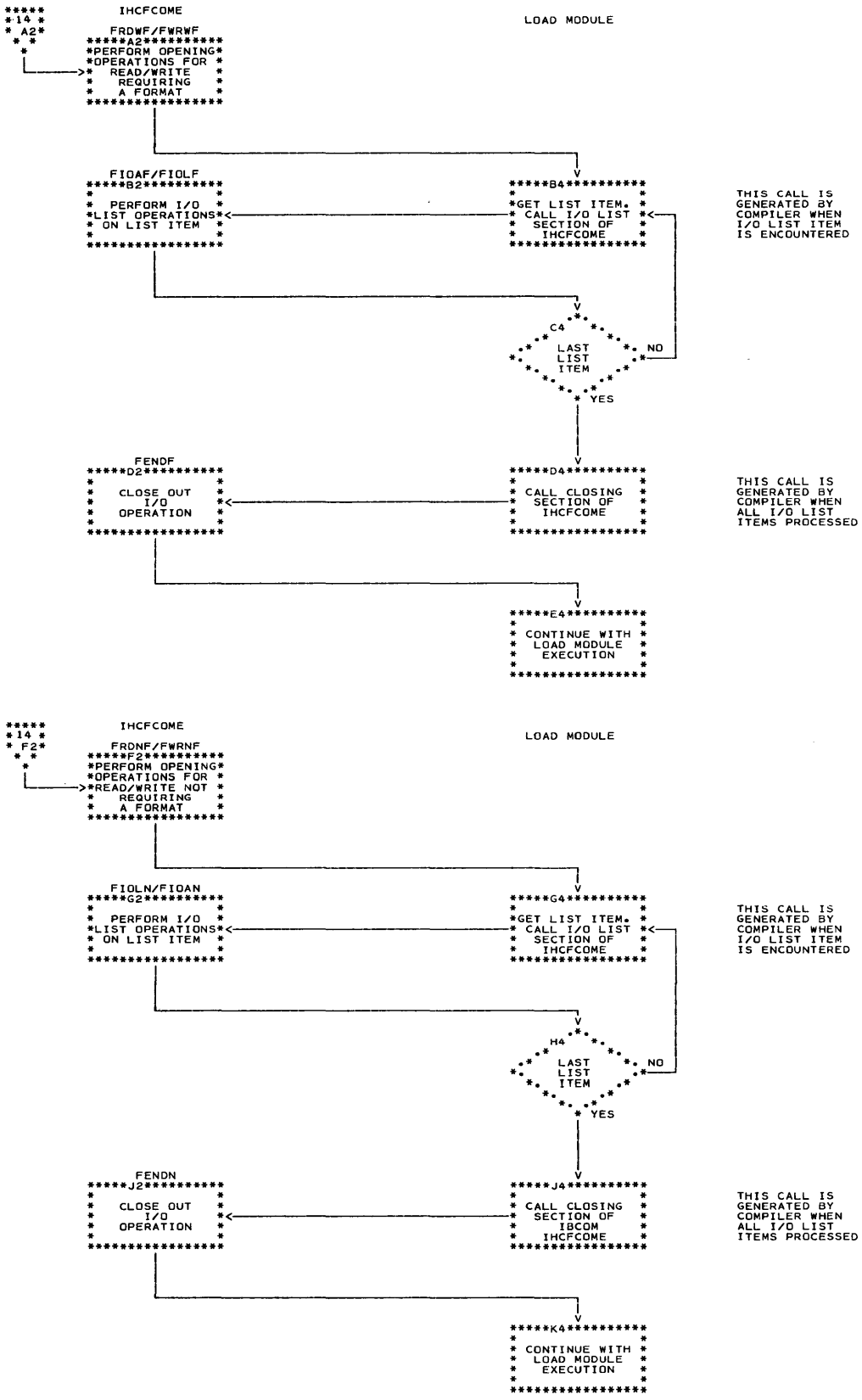
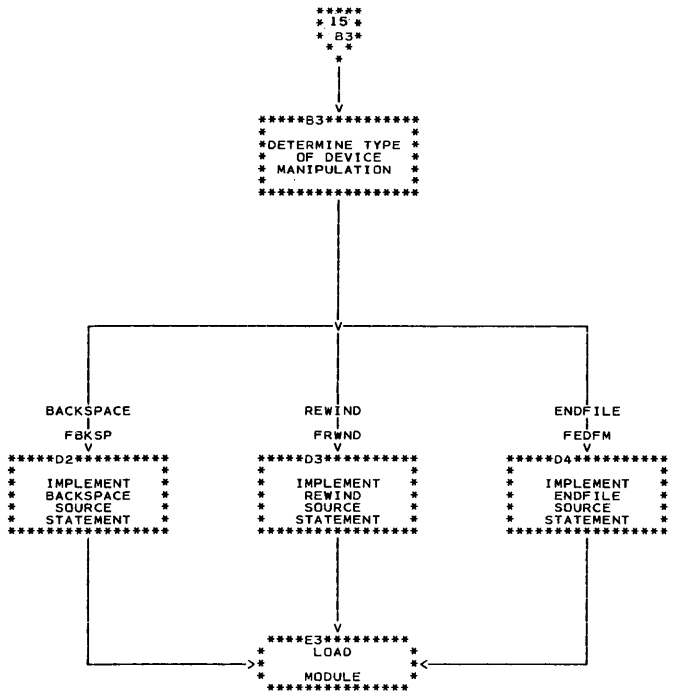


Chart 15. Device Manipulation and Write-to-Operator Routines



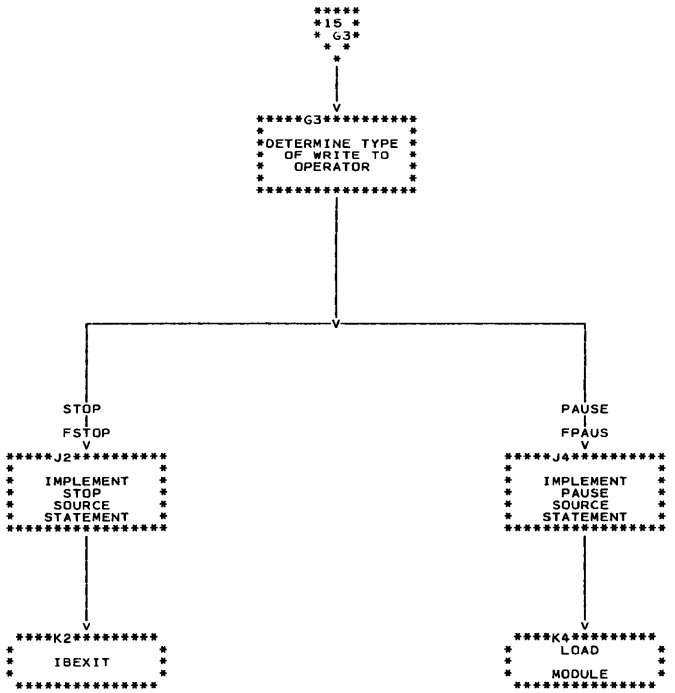


Table 26. IHCFCOME Routine/Subroutine Directory

Routine/Subroutine	Function
FBKSP	Implements the BACKSPACE source statement.
FCVAI	Reads alphameric data.
FCVAO	Writes alphameric 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.
FCVII	Reads integer data.
FCVIO	Writes integer 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 ENDFILE source statement.
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 the list variable of a READ or WRITE requiring a format.
FIOLN	I/O list section for the list variable of a READ or WRITE not requiring a format.
FPAUS	Implements the PAUSE source statement.
FRDNF	Opening section of a READ not requiring a format.
FRDWF	Opening section of a READ requiring a format.
FRWND	Implements the REWIND source statement.
FSTOP	Implements the STOP source statement.
FWRNF	Opening section for a WRITE not requiring a format.
FWRWF	Opening section for a WRITE requiring a format.
IBEXIT	Closes the data control blocks for all FORTRAN data sets that are still open and terminates the execution.
IBFERR	Processes object-time errors.
IBFINT	Processes arithmetic-type program interruptions.

IHCFIOSH

IHCFIOSH, the object-time FORTRAN input/output data management interface, receives input/output requests from IHCFCOME and submits them to the appropriate BSAM (basic sequential access method) routines and/or open and close routines for execution. Chart 16 illustrates the overall logic and the relationship among the routines of IHCFIOSH. Table 27, the IHCFIOSH routine directory, lists the routines and their functions.

Table and Blocks Used

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

Reserved	* n x 4 + 4	4 bytes
Data set reference number of error output device	Reserved	4 bytes
** Pointer to first unit block		4 bytes
.		.
.		.
.		.
** Pointer to last unit block		4 bytes
Default values for first unit block		8 bytes
.		.
.		.
.		.
Default values for last unit block		8 bytes

*n is the maximum number of units that can be referred to by the FORTRAN load module. The size of the unit table is equal to (8 + n x 12) bytes.

**The pointers to the various unit blocks are inserted into the unit assignment table when the unit blocks are constructed by IHCFIOSH.

UNIT ASSIGNMENT TABLE: The unit assignment table (IHCUATBL) resides on the FORTRAN system library (SYS1.FORTLIB). Its size depends on the maximum number of units that can be referred to during the execution of any FORTRAN load module. This number is specified during the system generation process via the FORTLIB macro-instruction. The unit assignment table is included (by the linkage editor) in the FORTRAN load module as a result of an external reference to it within IHCFIOSH.

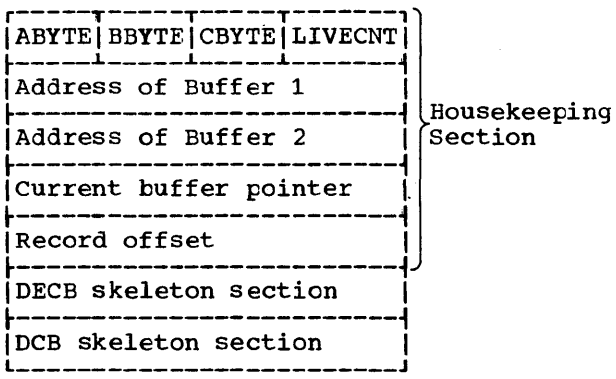
The unit assignment table has the following format:

The default value section of the unit assignment table contains standard values that IHCFIOSH inserts into the appropriate fields (e.g., BUFNO) of the DCB skeleton section of the unit block if the user either:

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

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

UNIT BLOCKS: The first reference to each unit number (data set reference number) by an input/output operation within the FORTRAN load module causes IHCFIOSH to construct a unit block for each unit number. The main storage for the unit blocks is obtained by IHCFIOSH via the GETMAIN macro-instruction. The addresses of the unit blocks are placed in the unit assignment table as the unit blocks are constructed. All subsequent references to the unit numbers are then made through the unit assignment block. Each unit block has the following format:



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

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

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

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

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

  Bit on

  0 - exit to IHCFCOME on I/O error
  1 - I/O error occurred
  2 - current buffer indicator
  3 - not used
  4 - end-of-current buffer indicator
  5 - blocked data set indicator
  6 - variable record format switch
  7 - not used

- CBYTE. This field also contains bits that are set and examined by IHCFIOSH. The bits and their meanings are as follows:

  Bit on

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

- LIVECNT. This field indicates whether any I/O operation performed for this data set is unchecked. (A value of 1 indicates that a previous read or write has not been checked; a value of 0 indicates that all previous read and write operations for this data set have been checked.)

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

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

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

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

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

Buffering

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

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

Communication With the Control Program

In requesting services of the control program, IHCFIOSH uses L and E forms of S-type macro-instructions (refer to the publication IBM System/360 Operating System: Control Program Services).

Operation

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

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

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

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

If the requested operation is that of write, a pointer to the buffer position, at which IHCFCOME 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 IHCFCOME.

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 IHCFCOME.

Previous Operation Read or Write: If the previous operation performed on the unit specified in the present I/O request was either a read or write, the initialization section determines the nature of the present I/O request. If it is a write, a pointer to the buffer position, at which

IHCFCOME 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 IHCFCOME.

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 IHCFCOME.

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

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

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

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

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

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

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

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

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

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

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

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

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

Chart 17 illustrates the execution-time I/O recovery procedure for any I/O errors detected by the I/O supervisor.

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

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

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

Write End-Of-Data Set: IHCFIOSH processes this request by issuing a CLOSE macro-instruction, Type = T. It then frees the I/O buffers by issuing a FREEPOOL macro-instruction, and returns control to IHCFCOME.

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

Chart 16. IHCFIOSH Overall Logic Diagram

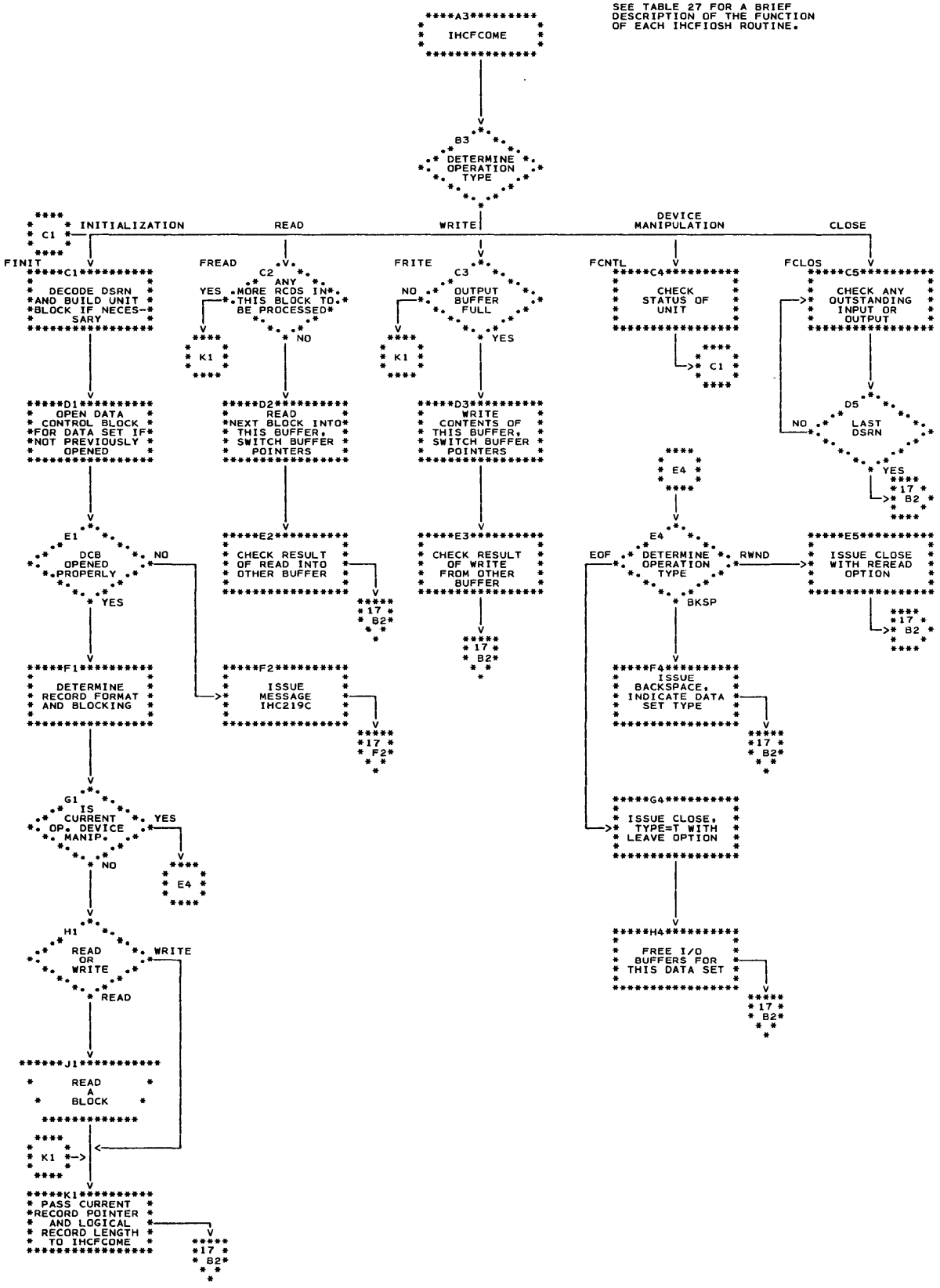


Chart 17. Execution-time I/O Recovery Procedure

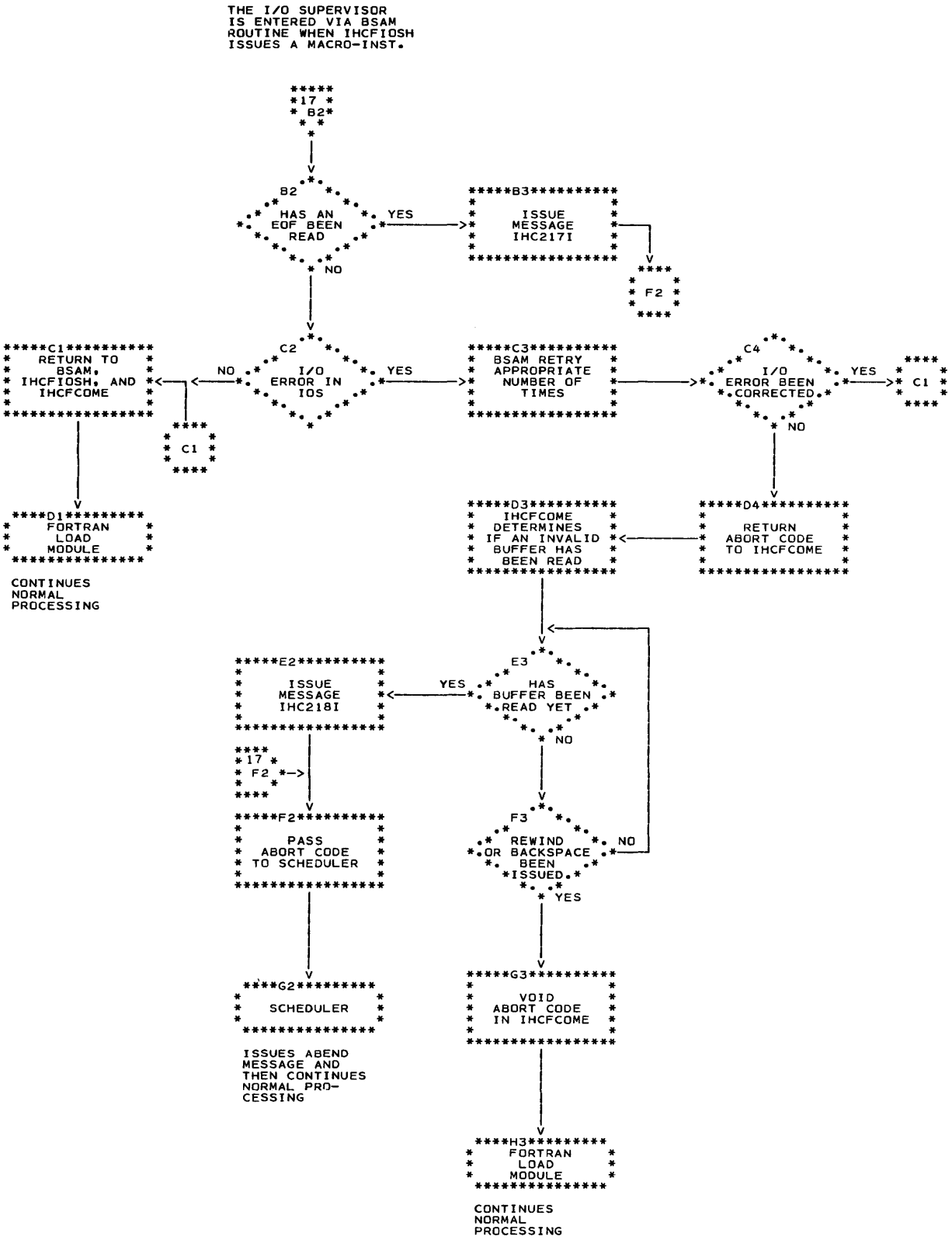


Table 27. IHCFIOSH Routine/Subroutine Directory

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

IHCIBERR

IHCIBERR, a member of the FORTRAN system library (SYS1.FORTLIB), processes object-time source statement errors if the LOAD option is specified. IHCIBERR is entered (via a calling sequence generated by Phase 20) when an internal sequence number (ISN) cannot be executed because of a source statement error.

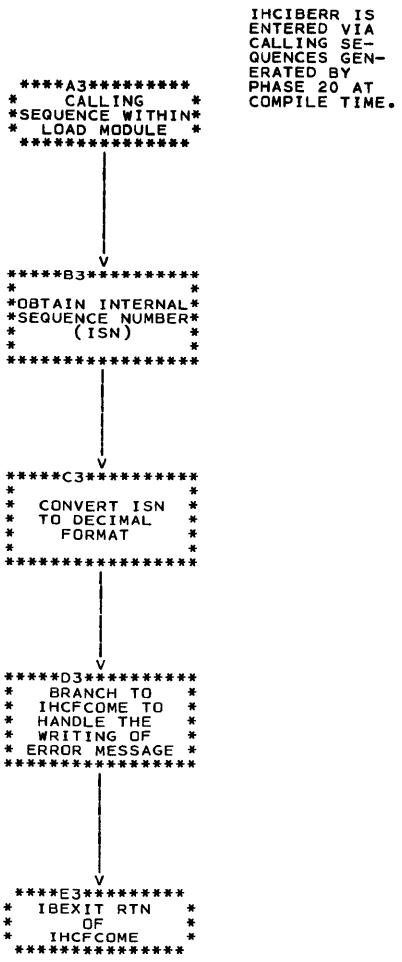
The ISN of the invalid source statement is obtained (from information in the calling sequence) and is then converted to decimal form. IHCIBERR then links to IHCFCOME to implement the writing of the following error message:

```
IHC230I - SOURCE ERROR AT ISN
          XXXX - EXECUTION FAILED
```

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

Chart 18 illustrates the overall logic of IHCIBERR.

Chart 18. IHCIBERR Overall Logic Diagram



LINKAGE TO THE INTERFACE MODULE

For SPACE compilations, the components of the compiler link to the interface module (IEJFAGA0) for:

- Input/output requests.
- End-of-phase/interlude requests.

In addition, for both SPACE and PRFRM compilations, the compiler components link to the interface module for patch requests and for print control operations.

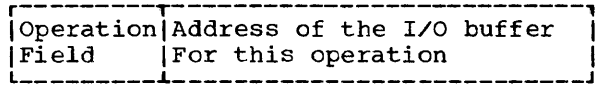
Input/Output Request Linkage

The linkage to the interface module for an I/O request is:

```
L      LNKREG,IOPARS
BAL    15,FIORTN
```

where

- INKREG is general register 0.
- IOPARS is the following 4-byte word:



Operation Field	Address of the I/O buffer For this operation
1 byte	3 bytes

The operation field bits and their meanings are illustrated in Table 28.

Table 28. Operation Field Bit Meanings

Bit 0	Check operation
Bit 1	Read operation
Bit 2	Write operation
Bit 3	Flush operation
Bit 4	Not used
Bits 5-7	000 - SYSIN is to be used 001 - SYSPUNCH is to be used 010 - SYSLIN is to be used 011 - SYSUT1 is to be used 100 - SYSUT2 is to be used 101 - SYSPRINT is to be used 110 - Not used 111 - Indicates that the address of the DECB to be used is supplied in PARREG,REG1.

- FIORTN is the name of a branch instruction in the communication area that branches to the I/O routine (SIORTN) of the interface module.

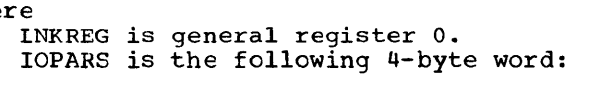
End-Of-Phase/Interlude Request Linkage

The linkage to the interface module for an end-of-phase/interlude condition is:

```
L      LNKREG,NXPARS
BC     15,FNEXT
```

where

- INKREG is general register 0.
- NXPARS is the following 4-byte word:



Entry point identifier of next phase/interlude	Data set disposition field
3 bytes	1 byte

The data set disposition field bits and their meanings are illustrated in Table 29.

Table 29. Data Set Disposition Field Bit Meanings

Bits 0-1	Not used
Bit 2	TCLOSE the DCB for SYSIN
Bit 3	TCLOSE the DCB for SYSPUNCH
Bit 4	TCLOSE the DCB for SYSLIN
Bit 5	TCLOSE the DCB for SYSUT1
Bit 6	TCLOSE the DCB for SYSUT2
Bit 7	TCLOSE the DCB for SYSPRINT

- FNEXT is the name of a branch instruction in the communication area that branches to the end-of-phase routine (SNEXT) of the interface module.

Patch Requests

The linkage to the interface module for a patch request is:

```
LR      WRKREG, BASEA
BAL     15, FPATCH
DC      C'XX'
```

where

- WRKREG is general register 14.
- BASEA contains the relative starting address of the component to be temporarily modified.
- FPATCH is the name of a branch instruction in the communication area that branches to the patch routine (PATCH) in the interface module.
- 'XX' is the fifth and sixth characters in the name of the component to be temporarily modified. (That is, 'XX' indicates the component to be modified.)

Print Control Operations

The linkage to the interface module for a print control operation is:

```
BAL     15, FPRTCTRL
DC      B'xxxxxxxx'
DC      AL3 (IOERR)
```

where

- FPRTCTRL is the name of a branch instruction in the communication area that branches to the print control operations routine (PRTCTRL) of the interface module.
- 'xxxxxxx' is the carriage control character.
- AL3 (IOERR) is an address constant containing the address of the I/O error routine of the component requesting the print control operation.

LINKAGE TO THE PERFORMANCE MODULE

For PRFRM compilations, the components of the compiler link to the performance module (IEJFAPA0) for:

- Input/output requests.
- End-of-phase requests.

Input/Output Request Linkage

The linkage to the performance module for an I/O request is basically the same as that described for the linkage to the interface module for an I/O request. The only difference is that the address in the branch and link (BAL) instruction is, in effect, replaced by the address of the I/O routine (PIORTN) of the performance module. The PIORTN routine, in turn, links to the I/O routine (SIORTN) of the interface module when it is either ready to read or write, or to check the result of a previous read or write.

End-Of-Phase Request Linkage

The linkage to the performance module for an end-of-phase request is basically the same as that described for the linkage to the interface module for an end-of-phase/interlude request. The only difference is that the address in the branch on condition (BC) instruction is, in effect, replaced by the address of the end-of-phase routine (PNEXT) of the performance module.

Note: Internally, the compiler components use symbolic names when transferring control to a subsequent component. The symbolic names and the actual names of the components are illustrated in Table 30.

Table 30. Symbolic and Actual Names of Compiler Components

Symbolic Name	Actual Name
IEJFAAA0*	Phase 1-Initial entry
IEJFAAB0	Phase 1-Subsequent entries
IEJFAGA0*	Interface module
IEJFAKA0	Print buffer module
IEJFAPA0*	Performance module
IEJFAXA0*	Source symbol module
IEJFEAA0	Phase 7
IEJFGAA0	Phase 10D
IEJFJAA0	Phase 10E
IEJFJGA0	Interlude 10E
IEJFLAA0	Phase 12
IEJFNAA0	Phase 14
IEJFNGA0	Interlude 14
IEJFPAA0	Phase 15
IEJFPGA0	Interlude 15
IEJFRAA0	Phase 20
IEJFVAA0	Phase 25
IEJFVCA0*	Object listing module
IEJFXAA0	Phase 30

*Never transferred to by another compiler component.

APPENDIX I: DIAGNOSTIC MESSAGES AND STATEMENT/EXPRESSION PROCESSING

This appendix contains the names of the phases and the routines within the phases that: (1) generate diagnostic messages, and (2) process the various FORTRAN statements and expressions.

DIAGNOSTIC MESSAGES

Two types of diagnostic messages are generated by the FORTRAN compiler - informative messages and error/warning messages. The messages produced by the compiler are explained in the IBM System/360 Operating System: FORTRAN IV (E) Programmer's Guide.

Informative Messages

Four informative messages are generated by the compiler to inform the programmer or operator of the status of the compilation. The messages and the phases and subroutines in which they are generated are illustrated in Table 31.

Table 31. Informative Messages

Message/number	Phase	Subroutine
IEJ001I	7	MESSGOUT
LEVEL: rmthyr OS/360 FORTRAN IV (E LEVEL SUBSET)COMPILATION DATE: yy.ddd	7	EJECTPRT
END OF COMPILATION	30	EOJOB
SIZE OF COMMON and SIZE OF OBJECT MODULE	30	ENDCRD

Error/Warning Messages

Each error/warning message produced by the compiler is identified by an associated number. Table 32 relates a message number with the phases and subroutines in which the corresponding message is generated.

Table 32. Error/Warning Messages

Message Number	Phase	Subroutine or Routine
IEJ002I	7	MESSGOUT
IEJ003I	7	MESSGOUT
IEJ004I	7	MESSGOUT
IEJ005I	7	MESSGOUT
IEJ006I	7	MESSGOUT
IEJ007I	7	MESSGOUT
IEJ008I	7	MESSGOUT
IEJ029I	10D	DIMSUB
IEJ030I	10D	COMMON, EQUIVP
IEJ031I	12	EQUIVP
IEJ032I	10D,10E	LITCON
IEJ033I	10D,10E	GETWD
IEJ034I	10D	FUNCT, SUBRUT
IEJ035I	10D	FUNCT, SUBRUT
IEJ036I	10E	ARITH
IEJ037I	10D,10E	CLASS, ARITH, ASF, IF
IEJ038I	10D	INTGER/REAL/DOUBLE, EXTERN, COMMON, EQUIV, DIM
IEJ039I	10D,10E	SYMTLU
IEJ041I	10D,10E	ASF, EXTERN, DIM
IEJ043I	10D,10E	INTGER/REAL/DOUBLE, GO
IEJ043I	12	ALOC
IEJ044I	10D,10E	LITCON
IEJ045I	10D,10E	LITCON
IEJ046I	10D,10E	LITCON

(Continued)

Table 32. Error/Warning Messages (Continued)

Message Number	Phase	Subroutine or Routine
IEJ047I	10D,10E	CLASS, DIM
IEJ048I	10D	DIMSUB
IEJ049I	10D	DIM, DIM90
IEJ050I	10D	EQUIV
IEJ051I	10D	EQUIV, DIM
IEJ051I	14	FCOMACHK
IEJ052I	10D	SUBS, EQUIV
IEJ053I	10D	SUBS
IEJ054I	10E	ASF
IEJ055I	10D	FUNC, SUBRUT
IEJ056I	10E	GO
IEJ057I	10E	READ/WRITE
IEJ058I	10E	READ/WRITE
IEJ060I	10D	EQUIV
IEJ061I	10D,10E	EOSR
IEJ063I	10E	EQUIV
IEJ064I	10D,10E,	LABTLU, SYMTLU,
IEJ064I	30	TWNFIV
IEJ065I	10D,10E	CLASS, LABLU, PAKNUM
IEJ066I	10E	DO
IEJ068I	10D,10E	LITCON
IEJ069I	10E	ASF
IEJ070I	10D	FUNCT, SUBRUT
IEJ071I	10E	CALL
IEJ072I	10E	ARITH
IEJ073I	10D,10E	PUTX
IEJ074I	10D	COMMON
IEJ075I	14	FORMAT, CKLM
IEJ076I	14	READ/READWR, FORMAT
IEJ077I	10D,10E	ASF, READ/WRITE, EOSR, DO, SUBS, EQUIV, FUNCT, SUBRUT, DIMSUB, DIM, SKPBLK
IEJ077I	14	READ/READWR, DO, FILLEG, SKPBLK
IEJ078I	14	CKENDO
IEJ079I	10E	GO
IEJ079I	14	READ/READWR, DO
IEJ080I	10E	GO
IEJ080I	14	READ/READWR
IEJ081I	10D,10E	ARITH, EQUIV
IEJ081I	14	READ/READWR, FMDCON, FMECON, FMFCON, FMTINT, FMACON, FORMAT
IEJ082I	10D,10E	LITCON
IEJ082I	14	NOFDCT, INTCON
IEJ083I	10D,10E	CSORN, INTCON
IEJ083I	14	INTCON
IEJ084I	10D,10E	WARN/ERRET
IEJ084I	14	ERROR, WARN
IEJ084I	15	ERROR, WARN
IEJ085I	12	DPALOC, SALO
IEJ085I	14	PRESCN
IEJ086I	14	BLANKZ
IEJ087I	14	FMDCON, FMECON, FMFCON, FMTINT, FMACON, FSUBST
IEJ088I	14	LPAREN
IEJ089I	14	UNITCK
IEJ090I	14	FQUOTE
IEJ091I	14	FMINUS, FPLUS
IEJ092I	14	FCOMMA
IEJ094I	14	FMDCON, FMECON, FMFCON, FMTINT, FMACON
IEJ095I	14	READ/READWR
IEJ096I	14	READ/READWR
IEJ097I	14	INSAV
IEJ098I	14	FQUOTE

Table 32. Error/Warning Messages

Message Number	Phase	Subroutine or Routine
IEJ099I	14	FQUOTE
IEJ100I	14	DO, READ/READWR
IEJ123I	15	MOPUP
IEJ124I	15	COMMA
IEJ125I	15	DO, BEGIO
IEJ126I	15	CKARG
IEJ127I	12	COMALO, ALOC
IEJ127I	15	PRESCN, UMINUS, UPLUS, FOSCAN
IEJ128I	15	LFTPRN
IEJ129I	15	TYPE
IEJ130I	15	COMMA
IEJ131I	15	INLIN1
IEJ132I	15	LABEL
IEJ133I	15	EQUALS
IEJ135I	15	COMMA, TYPE
IEJ136I	15	LAB
IEJ137I	15	COMMA, TYPE, RTPRN, FOSCAN
IEJ139I	15	COMMA
IEJ140I	15	FOSCAN
IEJ141I	15	COMMA
IEJ142I	15	DO, BEGIO
IEJ143I	15	EQUALS
IEJ144I	15	ARTHIF
IEJ142I	12	EXTCOM
IEJ145I	20	PHEND
IEJ143I	12	COMALO, RENTER/ENTER, SWROOT
IEJ147I	12	EQUIVP

(Continued)

Message Number	Phase	Subroutine or Routine
IEJ148I	12	RENTER/ENTER, SWROOT
IEJ149I	12	COMALO
IEJ150I	12	ALOC
IEJ159I	15	MOPUP
IEJ160I	14	INTCON
IEJ160I	15	COMMA
IEJ161I	12	EXTCOM
IEJ162I	10D,10E	CLASS
IEJ163I	10D,10E	LITCON
IEJ164I	10E	CONT/RETURN
IEJ164I	14	FORMAT
IEJ166I	10D,10E	EOSR, DO, FUNCT,SUBRUT
IEJ166I	14	READ/READWR
IEJ167I	14	LINECK
IEJ168I	10D,10E	EOSR
IEJ169I	10D	DIMSUB
IEJ169I	15	COMMA
IEJ171I	10D,10E	EOSR
IEJ171I	14	RPAREN
IEJ172I	10E	ASF
IEJ173I	10E	ARITH
IEJ174I	15	EQUALS, LFTPRN, INARG, TYPE
IEJ175I	14	LABEL

STATEMENT/EXPRESSION PROCESSING

Table 33 indicates the routine/ subroutine responsible for the processing of the statement/expression under consideration, and the phase in which it appears.

Table 33. Statement/Expression Processing

Statement/ Expression	Phase 10D/10E	Phase 12	Phase 14	Phase 15	Phase 20	Phase 25	Phase 30
Arithmetic Expression or statement	ARITH (E)		PASSON	FOSCAN	ARITH	RXGEN	
FUNCTION Call	ARITH (E)	LDCN	PASSON	FOSCAN	CALSEQ	FUNGEN/ EREXIT	
Subscripted Variable	SUBS (E)	SSCK	PASSON	FOSCAN, MVSBXX/ MVSBRX	SUBVP	SAOP, AOP	
SF definition and expansion	ASF (E)	LDCN	ASF	FOSCAN	ARITH	ASFDEF, ASFEXP	
Statement Number Definitions	CLASS (E)	ASSNBL	LABEL	LABEL	LABEL	LABEL	
SF Call	ARITH (E)	LDCN	PASSON	FOSCAN	CALSEQ	ASFUSE	
BACKSPACE	BKSP/REWIND END/ENDFIL (E)		BSPREF	D02	ESDRLD	RDWRT	
CALL	CALL (E)	LDCN	PASSON	FOSCAN	CALSEQ, IFCALL	FUNGEN/ EREXIT	
COMMON	COMMON (D)	COMAL					
Computed GOTO	GO (E)		CGOTO	CGOTO	COGOTO	CGOTO	
CONTINUE	CONT RETURN (E)		SKIP	SKIP			
DIMENSION	DIM (D)						
DO	DO (E)		DO	DO	DO	DO1, ENDDO	
DOUBLE PRECISION	INTGER/ READ/DOUBLE (D)	DPALOC					
END	BKSP/REWIND/ END/ENDFIL (E)		END	MOPUP	PHEND	END	ENDCRD
ENDFILE	BKSP/REWIND/ END/ENDFIL (E)		BSPREF	D02	ESDRLD	RDWRT	
EQUIVALENCE	EQUIV (D)	EQUIVP					
EXTERNAL	EXTERN (D)	LDCN					
FORMAT	FORMAT (D,E)		FORMAT				
FUNCTION	FUNCT/SUBRUT (D)	LDCN	SUBFUN	FHDR		SUBRUT	
GO	GO (E)		ENDOCK	GOTO		TRGEN	
IF	IF (E)		ENDOCK	FOSCAN	IFCALL	ARITHI	
In-line Functions	ARITH (E)	LDCN	PASSON	FOSCAN	CKCOD	FUNGEN/ EREXIT	

(Continued)

Table 33. Statement/Expression Processing (Continued)

Statement/ Expression	Phase 10D/10E	Phase 12	Phase 14	Phase 15	Phase 20	Phase 25	Phase 30
INTEGER	INTGER/ REAL/DOUBLE (D)	SALO					
PAUSE	STOP/PAUSE (E)		PAUSE	DO2		STOP/PAUSE	
READ	READ/WRITE (E)		READ	DO2	READ, LIST	RDWRT/ IOLIST	
REAL	INTGER/ REAL/DOUBLE (D)	SALO					
RETURN	CONT RETURN (E)		RETURN	SKIP		RETURN	
REWIND	BKSP/REWIND/ END/ENDFIL (E)		BSPREF	DO2	ESDRLD	RDWRT	
STOP	STOP/PAUSE (E)		STOP	DO2		STOP/PAUSE	
SUBROUTINE	FUNCT/SUBRUT (D)	LDCN	SUBFUN	DO2		SUBRUT	
WRITE	READ/WRITE (E)		READWR	DO2	LIST	RDWRT/ IOLIST	

The amount of main storage allocated to the compiler depends on whether a SPACE or a PRFRM compilation is being performed.

FOR SPACE COMPILATIONS

For SPACE compilations, the compiler requires main storage for:

- Load modules (phases, interludes, print buffer, and interface).
- Resident tables (dictionary, overflow table, SEGMAL).
- Internal text buffers.
- BSAM I/O routines.

The main storage required by each phase/interlude of the compiler need be contiguous only for each control section. Figures 47 through 53 reflect the main storage allocation associated with each successive phase/interlude as it performs its functions, when only a minimal amount of storage (15K bytes, where K = 1024) is available for compilation.

When the main storage allocated to the compiler (specified in the SIZE option) is greater than 15K bytes, the internal text buffers may be interspersed within the area occupied by the dictionary and the overflow table. In this case, there need be no relationship between the various areas required by the compiler.

These figures are schematics showing the main storage allocated; proportional sizes within the diagrams do not necessarily indicate proportional amounts of main storage.

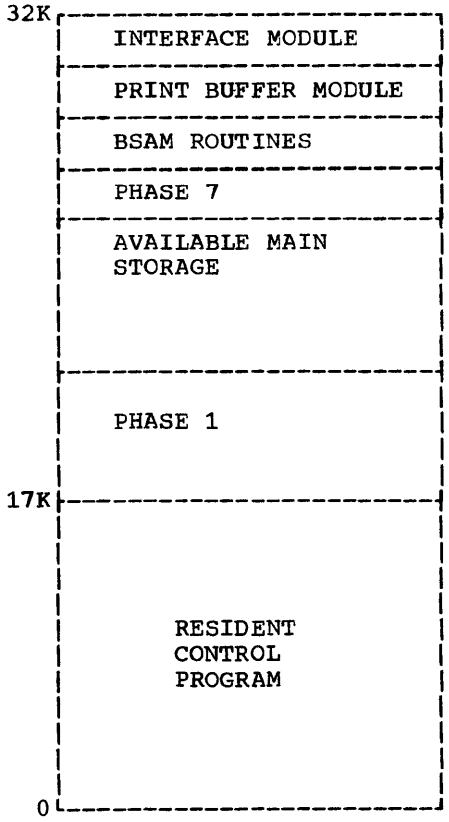


Figure 47. End of Phase 1 (initial entry)

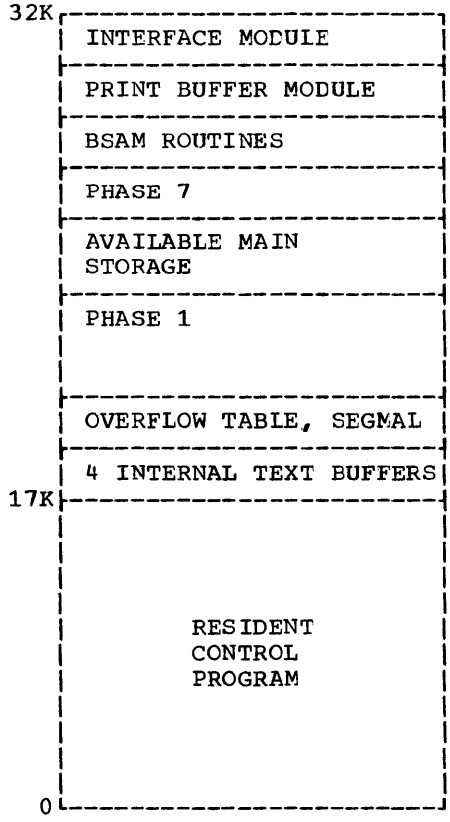


Figure 48. End of Phase 1 (subsequent entries)

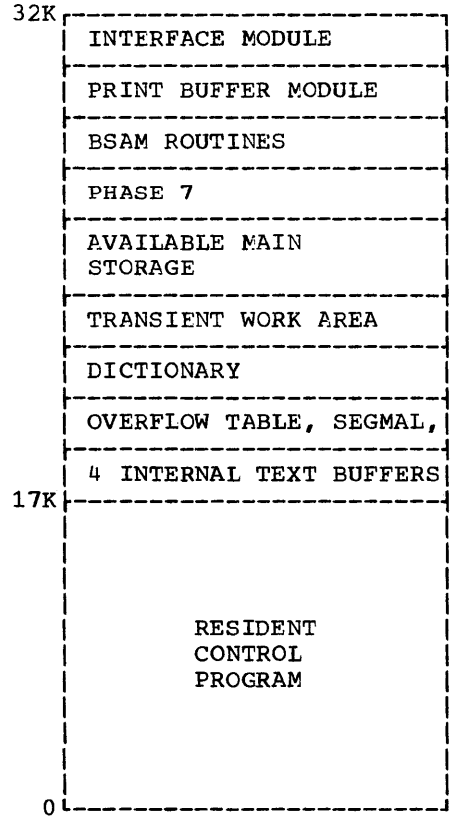


Figure 49. End of Phase 7

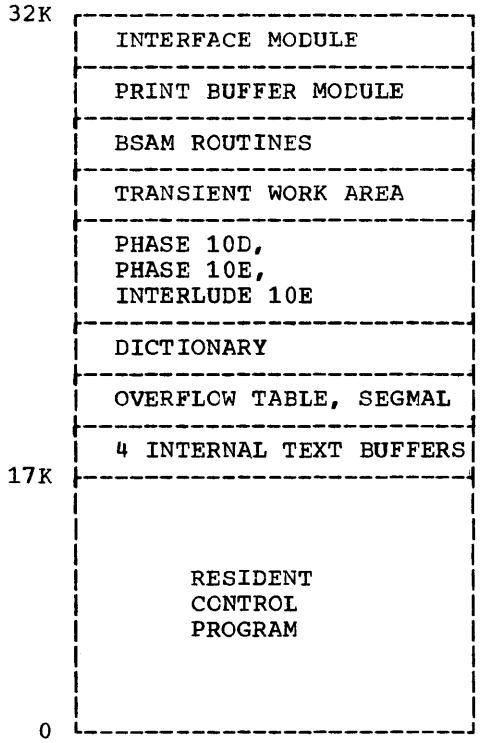


Figure 50. Phases 10D and 10E, and Interlude 10E

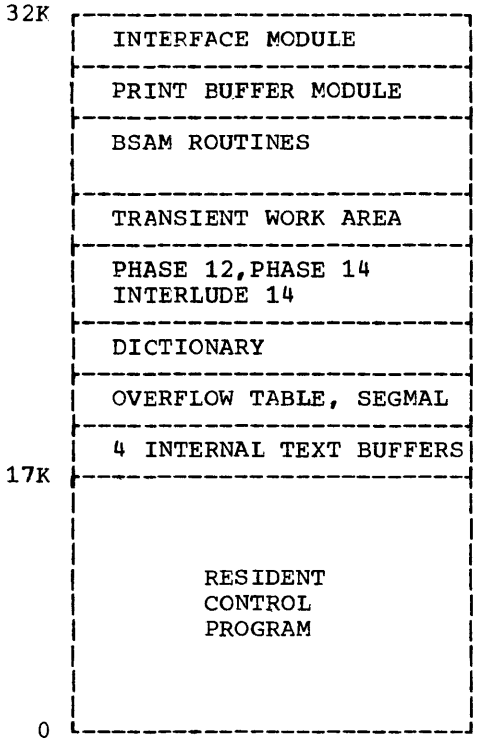


Figure 51. Phases 12 and 14, and Interlude 14

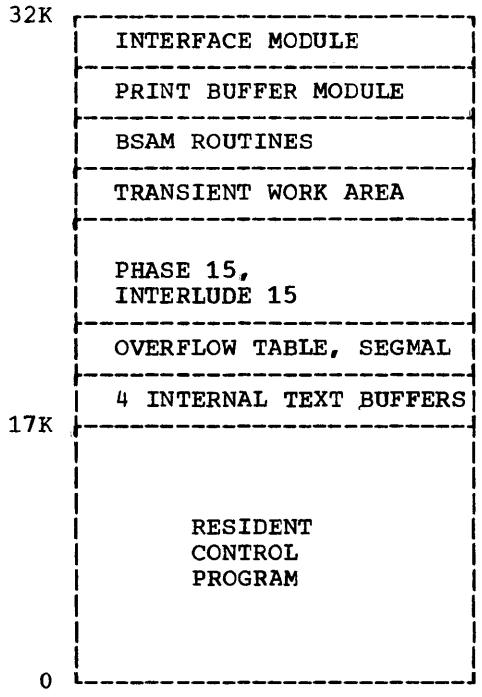


Figure 52. Phase 15 and Interlude 15

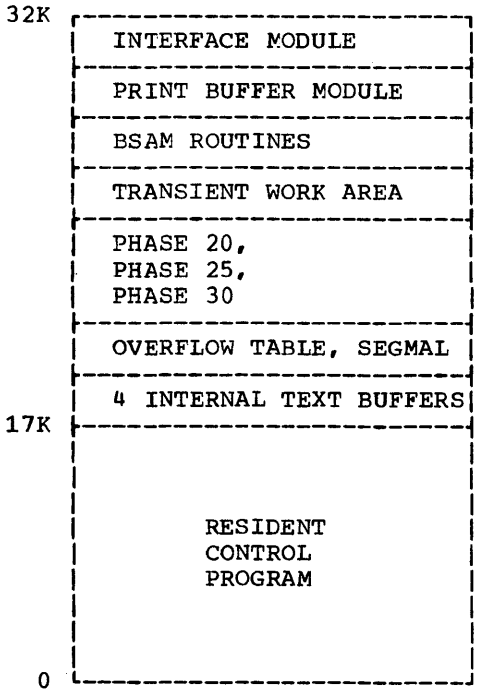


Figure 53. Phases 20, 25, and 30

FOR PRFRM COMPILATIONS

For PRFRM compilations, the compiler requires main storage for:

- Load modules (phases, interface, print buffer, and performance).
- Resident tables (dictionary, overflow table, and SEGMAL).
- Internal text buffers.
- BSAM I/O routines.
- Block/deblock buffers if blocking is specified.

The main storage required by any given phase of the compiler need be contiguous only for each control section within that phase. Figure 54 reflects the main storage allocation for the duration of a PRFRM compilation, when only a minimal amount of main storage (19K bytes, where K=1024) is available for compilation.

When the main storage allocated to the compiler (specified in the SIZE option) is greater than 19K bytes, the internal text buffers may be interspersed within the area occupied by the dictionary and the overflow table. In this case, there need be no relationship among the various areas required by the compiler.

Figure 54 is a schematic showing the main storage allocated; proportional sizes within the diagram do not necessarily indicate proportional amounts of main storage.

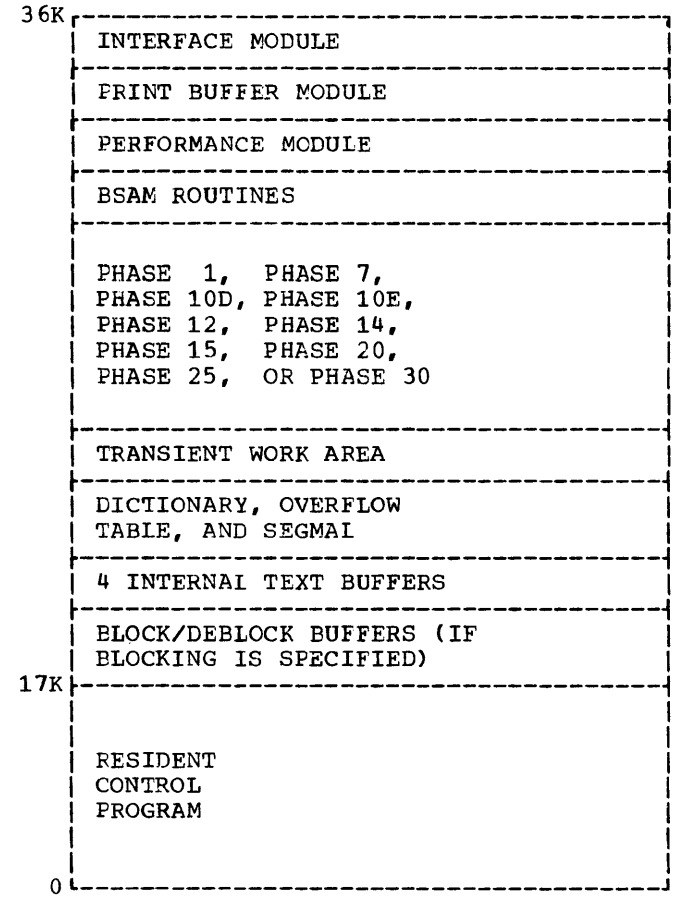


Figure 54. Main Storage Allocation for a PRFRM Compilation

The communication area is a central gathering area used to communicate necessary information between the various phases of the compiler. The communication area, as a portion of the interface module, is resident throughout the compilation.

Various bits in the communication area are examined by the phases of the compiler. The status of these bits determines the following:

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

If the bit in question is a 0, the option has not been specified, or the action is not to be taken; if the bit is a 1, the option has been specified, or the action is to be taken.

Several entries in the communication area are equated to the addresses of other entries in the communication area used during earlier phases. Equating the entries keeps the size of the communication area to a minimum.

The communication area is assembled as a DSECT (dummy section) within each phase. This allows the phases to symbolically address the entries in the communication area without the communication area actually residing in each phase.

Table 34 indicates the format and organization of the communication area.

Table 34. Communication Area

Entry	Size	Meaning	
FCOMM	DS XL4	BIT0	SOURCE *
		BIT1	DECK *
		BIT2	MAP *
		BIT3	ADJUST *
		BIT4	PRFRM *
		BITS	5-6 00 NOLOAD*
			11 LOAD *
		BIT7	BCD VERSION OF SCODE REQUESTED *
		BIT8	NAME PARAMETER EXISTED
		BITS	9-10 00 MAIN PROGRAM
			10 SUBROUTINE SUBPROGRAM
			11 FUNCTION SUBPROGRAM
		BIT11	FUNCTION NAME DEFINED
		BIT12	OBJECT MODULE CALLS AN EXTERNAL S/P
		BIT13	COMMON AND EQUIVALENCE TEXT ALL IN STORAGE
		BIT14	LAST COMPILE OF THIS JOB STEP-PH 10E/1
		BIT15	ERROR IN ANY COMPILE OF A BATCH RUN
		BIT16	WARNING MESSAGES
		BIT17	ERROR MESSAGES
		BIT18	MESSAGE IN CURRENT STATEMENT-PH 10D/10E
		BIT19	WARNING IN ANY COMPILE OF A BATCH RUN
		BIT20	ABORT COMPILATION
		BIT21	ALL INTERNAL TEXT IN STORAGE
		BIT22	ONE INTERNAL TEXT RECORD-PH 10D/10E
			OBJ. MOD. USES A SPILL BASE REG-PH 12/25
			BRANCH LIST TEXT NOT ALL IN STORAGE-PH 25/30
		BIT23	OBJECT LISTING
		BIT24	OTHER THAN FIRST COMPILE
		BIT25	COMPILATION RESTARTED
		BIT26	INVALID OPTION(S) IN 'PARM' FIELD
		BIT27	'NAME' OPTION TOO LONG-TRUNCATED
		BITS28-31	SPARE

(Continued)

Table 34. Communication Area (Continued)

Entry	Size	Meaning	
FSIZE	DS F	BYTES OF STORAGE REQUESTED FOR COMPILER*	
FDATE	DS CL5	YEAR (2 DIGITS), DAY (3 DIGITS)	
FLINELNG	DS X	OBJECT PROGRAM PRINT LINE LENGTH *	
FINDEX	DS H	DISPLACEMENT FROM FCOMM TO FDECBIN	
FMAXLINE	DS H	MAXIMUM NUMBER OF LINES CN LISTING PAGE	
FCURLINE	DS H	CURRENT LINE ON LISTING PAGE	
FIEJF	DS CL4	FORTRAN E INTERNAL COMPONENT CODE - IEJF	
FPHASE	DS CL4	ENTRY POINT OF PHASE IN CCNTRCL	
FDMRRDCD	DS X	HI-ORDER BYTE OF REREAD ITEM IN CLOSE LIST	
FDMLSTCD	DS X	HI-ORDER BYTE OF LAST ITEM IN CLOSE LIST	
FPRTCTRL	DS 2H	BRANCH TO PRINT CONTROL ROUTINE	
THE CONTENTS OF THE NEXT 4 FIELDS DEPENDS ON WHETHER A SPACE OR A PRFRM COMPILATION IS BEING PERFORMED.		FOR SPACE COMPILATIONS -	FOR PRFRM COMPILATIONS -
FIORTN	DS 2H	B SIORTN	MVI FPRFRMDL,X'4'
FNEXT	DS 2H	B SNEXT	L 13,FPRFRMDL
	DS H	(NOT USED)	BR 13
FPRFRMDL	DS A	ZERO	ADDR. OF IEJFAPA0
FAGA0END	DS A	ADDRESS OF (END OF INTERFACE MODULE + ONE)	
FSAVADDR	DS A	ADDRESS OF CONTROL PROGRAM SAVE AREA	
FTXTBFSZ	DS H	SIZE OF INTERNAL TEXT BUFFERS	
FTXTPTR	DS H	ADDR. OF NEXT INT. TEXT RCD.-PH. 10D/E,12/14	
FTXTBFA1	DS A	ADDRESS OF INTERNAL TEXT BUFFER 1 - SYSUT1	
FTXTBFA2	DS A	ADDRESS OF INTERNAL TEXT BUFFER 2 - SYSUT1	
FTXTBFB1	DS A	ADDRESS OF INTERNAL TEXT BUFFER 1 - SYSUT2	
FTXTBFB2	DS A	ADDRESS OF INTERNAL TEXT BUFFER 2 - SYSUT2	
FPRTBUF1	DS A	ADDRESS OF FIRST PRINT BUFFER - PHASE 1/14	
FPRTBUF2	DS A	ADDRESS OF SECOND PRINT BUFFER - PHASE 1/14	
FINITBFS	DS 4A	INITIAL TEXT BUFFER POINTERS	
FDICTNDX	DS A	ADDRESS OF DICTIONARY INDEX - PHASE 7/12	
FOVFLNDX	DS A	ADDRESS OF OVERFLOW INDEX	
FDICTBLK	DS A	DICT. BLOCK NOW BEING BUILT - PH. 10D/E	
FOVFLBLK	DS A	OVFL. BLOCK NOW BEING BUILT - PH. 10D/E	
FDICTNXT	DS A	DICT. ENTRY NEXT TO BE BUILT - PH. 10D/E	
FOVFLNXT	DS A	OVFL. ENTRY NEXT TO BE BUILT - PH. 10D/14	
FISNEX1	DS F	ISN OF FIRST EXECUTABLE-PHASE 10D/E	
FOBJPROG	DS CL6	NAME OF OBJECT PROGRAM	
FOBJREGS	DS X	BIT 3, EXTERNAL FUNCTION HAS BEEN CALLED BITS 4-7, LOWEST INDEX REGISTER IN OBJ. PROG.	
FASFCNT	DS X	COUNT OF SF'S IN OBJECT PROGRAM	
FDOCOUNT	DS H	NUMBER OF DO STATEMENTS	
	DS H	SPARE	

(Continued)

Table 34. Communication Area (Continued)

FCOMSIZE	EQU FDICTBLK	SIZE OF OBJECT PROGRAM COMMON - PH. 12/25
FALSIZE	EQU FDICTBLK+2	SIZE OF OBJ. PROG. ARGUMENT LIST - PH. 15/20
FBLSIZE	EQU FOVFLBLK	SIZE OF OBJ. PROG. BRANCH LIST - PH. 12/25
FBLSTRT	EQU FOVFLBLK+2	ADDR. OF OBJ. PROG. BRANCH LIST - PH. 12/25
FASFDOBL	EQU FOVFLNXT+2	ADDRESS OF ASF/DO BRANCH LIST - PH. 20/25
FBVSTRT	EQU FDICTNXT	ADDR. OF OBJ. PROG. BASE VAL. LIST - PH. 12/25
FOBJSTRT	EQU FDICTNXT+2	STARTING ADDR. OF OBJECT PROGRAM - PH. 12/25
FLOCCTR	EQU FISNEX1	LOCATION COUNTER FOR OBJ. PROG. - PH. 12/25
FFNCADDR	EQU FDICTBLK+2	ADDRESS OF RESULT (FUNCTION S/P) - PH. 14/15
FIBCOM	EQU FOVFLNXT	ADDRESS OF IBCOM - PHASE 20/25
FOBJERR	EQU FDICTBLK+2	ADDR. OF OBJ. PROG. ERROR RTNE. - PH. 20/25
FDECKSEQ	EQU FDICTNDX	OBJECT PROGRAM DECK SEQUENCE NUMBER - PH. 12/25
FESDSEQ	EQU FDICTNDX+2	OBJECT PROGRAM ESD SEQUENCE NUMBER - PH. 12/20
FALSTRT	DS F	DSRN ARGUMENT LIST ADDRESS
FDATEMP	DS F	ADDRESS OF DIRECT ACCESS I/O TEMPORARY AREA
FDEFILCT	DS F	'DEFINE FILE' DSRN COUNT - PH. 10D/20
FDIOCS	EQU FDEFILCT	ADDRESS OF DIOCS - PH. 20/25
FPATCH	DS 2H	BRANCH TO PATCH ROUTINE IN INTERFACE MODULE
FPTCHTBL	DS A	ADDRESS OF PATCH TABLE
FPTCHPTR	DS A	PATCH TABLE ENTRY NEXT TO BE POSTED
FSORSYM1	DS A	ADDRESS OF SORSYM TABLE
FSORSYM2	DS A	SORSYM TABLE ENTRY NEXT TO BE BUILT

*Default values for these compiler options may be specified by the user during the system generation process via the FORTRAN macro-instruction. The default values are assumed if the corresponding parameters in the PARM field of the user's EXEC statement are not included.

Phase 10D and Phase 10E convert each FORTRAN source statement into a form (intermediate text) usable to subsequent phases of the compiler. Intermediate text is developed by scanning the source statements from left-to-right and by constructing one-word intermediate text entries for the source text contained in the statements.

Phase 10D scans the declarative statements in the source module, and creates intermediate text for those statements. When Phase 10D encounters either the first statement function or the first executable statement, control is passed to Phase 10E via the interface module. Phase 10E continues the scan of the source module and creates intermediate text for statement functions and executable statements.

As source statements are scanned, entries are made to the dictionary and overflow table. The information in the dictionary and overflow table supplements the intermediate text in the generation of machine language instructions by subsequent phases of the compiler. This information is associated with the intermediate text entries by means of pointers that reside in the text entries.

Each source statement of the source module consists of one or more card images. To scan source statements, each card image of the source module is first read into one of two I/O buffers in the print buffer module (IEJFAKAO). The double-buffer scheme allows for overlapping the scanning of a card image in one buffer with the reading of the next card image of the source module into the other buffer. If the SOURCE option is specified, the I/O buffers are used to print a listing of the source module.

In general, the processing of a source statement is divided into three operations:

- Preliminary scan of the card image(s) for the statement.
- Classification scan of the first card image for the statement.
- Reserved word or arithmetic scan of the card image(s) for the statement, which scans the source text of the statement. (The reserved word or arithmetic scan also creates intermediate text.)

PRELIMINARY SCAN

The preliminary scan first determines the address of the end of the source text in the card image to be processed. This address is obtained by examining the card image from right-to-left in groups of four bytes. The address of the last blank group encountered is used as the ending address of the card image. This address is used in the reserved word or arithmetic scan of the card image and indicates the point at which the scan of the card image and the creation of intermediate text for the card image is to terminate. In the case of the last card image for a statement, the ending address indicates the end of the statement.

The preliminary scan then determines the type of the card image to be scanned. A card image may correspond to the start of a FORTRAN statement, the continuation of a FORTRAN statement, or a user's comment.

If the card image corresponds to the start of a FORTRAN statement, a unique internal statement number is assigned to the statement. This number is placed in front of the card image in the buffer containing that card image. Control is then passed to the classification scan.

If the card image corresponds to a continuation of a FORTRAN statement, a new internal statement number is not assigned. Control is immediately passed to the classification scan.

If the card image corresponds to a user's comment, no further processing is required. The next card image of the source module is read into the buffer that contained the comments card image. The address of the other buffer (previously filled) is obtained from the communication area, and scanning starts for the card image in that buffer.

In each case, if the SOURCE option is specified the buffer containing the card image is first written onto the SYSPRINT data set before any further processing.

CLASSIFICATION SCAN

The classification scan determines the type (arithmetic or reserved word) of the FORTRAN statement to be processed. The

first action taken by the classification scan is to determine if a statement number defines the statement under consideration. If a statement number is associated with the statement, an overflow table entry for that statement number is created.

The next item of the source statement is then obtained. If the item is a symbol, control is passed to a routine that scans arithmetic statements. If the item is a reserved word (e.g., READ), control is passed to the appropriate reserved word routine. The arithmetic or reserved word routine controls the scanning of the remainder of the statement, and creates intermediate text for the statement.

If the item is neither a symbol nor a reserved word, the source statement in question is invalid. Processing of that statement is terminated, and processing of the next statement of the source module begins.

RESERVED WORD OR ARITHMETIC SCAN

The main function of the reserved word or arithmetic scan is to scan the card image(s) for each statement of the source module. During this scan, dictionary and overflow table entries are constructed, and intermediate text entries are created. In addition, each statement is examined for correct use of the FORTRAN IV (E) language.

The reserved word or arithmetic scan is performed by either a reserved word routine or the arithmetic routine. A reserved word routine exists for each of the reserved word source statements. Certain reserved word routines, namely those that process statements that may contain arithmetic expressions (e.g., IF and CALL statements) and those that process statements that contain I/O lists (e.g., READ and WRITE statements) pass control to the arithmetic routine to complete the scanning of the associated reserved word statements.

When the appropriate reserved word routine or the arithmetic routine receives control, a left-to-right scan of the current card image is then initiated. The first operand of the card image is obtained, and a check is made to determine if a dictionary or overflow table entry has previously been created for the operand. If an entry has not been created, a dictionary or overflow table entry (depending on the operand) is created and entered in the appropriate resident table. Scanning is resumed and the first operator of the card image is obtained.

The intermediate text for each card image is developed by constructing intermediate text entries for operator-operand pairs as they are scanned by a reserved word routine or the arithmetic routine. In this context, operator refers to commas, parentheses, etc., as well as to arithmetic operations (e.g., + and -). Operand refers to variables, constants, statement numbers, data set reference numbers, etc., that are operated on.

The procedure of: (1) scanning operators and operands, (2) constructing dictionary or overflow table entries when necessary for the operands, and (3) developing intermediate text entries for the operator-operand pairs is repeated until the end of the card image is recognized by the reserved word or arithmetic scan.

When the address indicating the end of the card image is recognized by the reserved word or arithmetic scan, the next card image of the source module is read into the buffer that contained the card image just processed. The address of the other buffer (previously filled) is obtained from the communication area, and processing starts for the card image in that buffer.

When an entire source statement has been scanned, a special intermediate text entry indicating the end of the intermediate text representation for a given statement is generated and then written onto an intermediate storage data set at the end of the intermediate text representation for the statement. This special text entry contains the internal statement number assigned to the statement by the preliminary scan section.

During the reserved word or arithmetic scan, each card image is examined for proper use of the FORTRAN IV (E) language. The format of the card image is checked to see if the statement associated with the card image has been coded properly by the source programmer.

If a serious error is encountered, scanning of the statement associated with the card image is terminated. An intermediate text word indicating the end of the intermediate text representation for the statement is generated and then written onto an intermediate storage data set. This text word also indicates that an error was encountered in the processing of the statement. An intermediate text word, representing the error, which contains a number corresponding to the specific error detected, is generated and then written onto the intermediate storage data set at the end of the intermediate text represen-

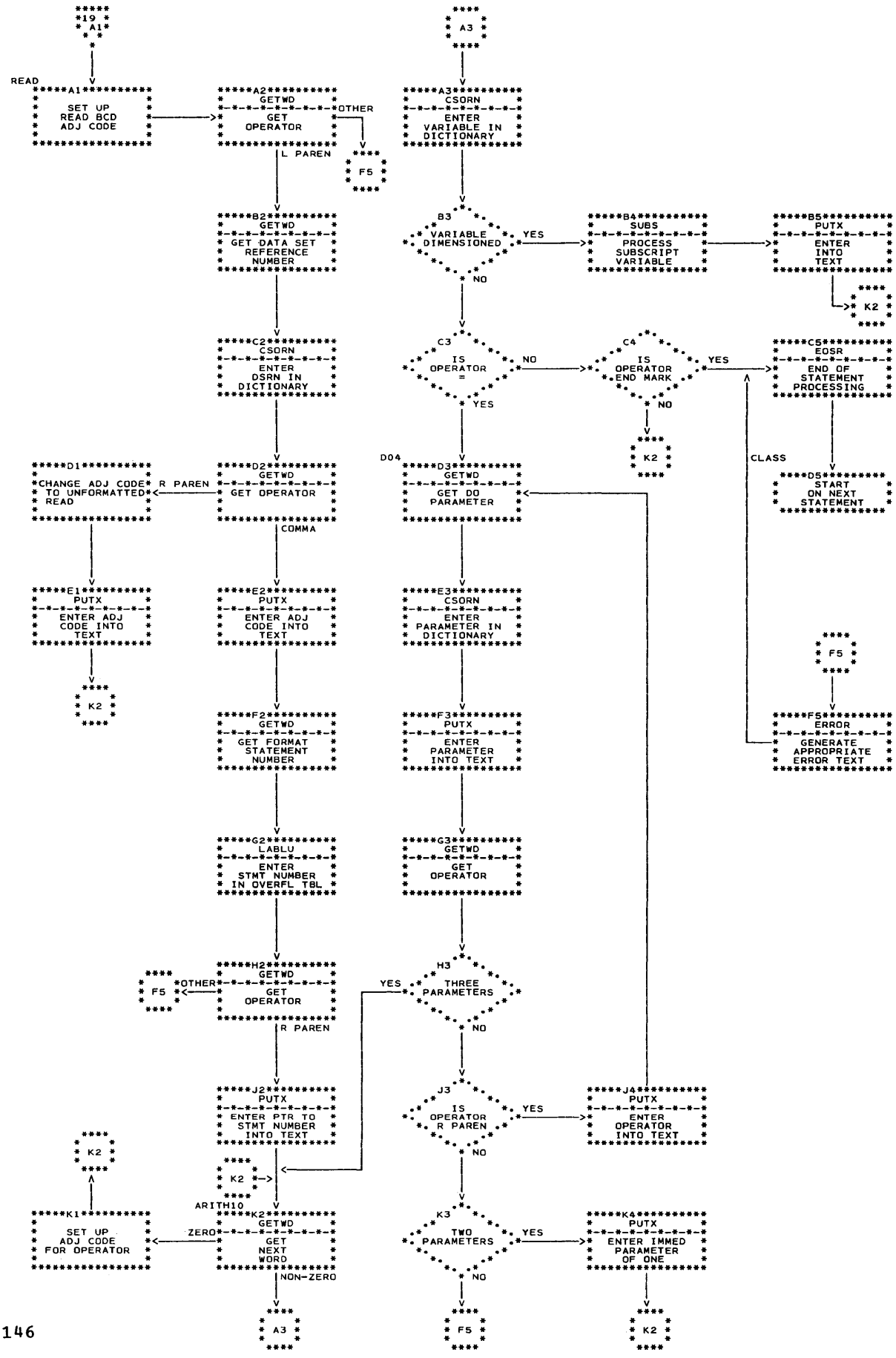
tation for the statement in which the error was detected.

If an error is encountered that is not serious enough to terminate the scan of a statement, an intermediate text word representing a warning is generated. This word is saved and scanning is resumed. When the scan of the statement is terminated (either when the end of the statement is recognized or when a serious error is encountered), the warning text word is written onto the intermediate storage data set immediately following the text word that indicates the end of the intermediate text representation for the statement and any intermediate text words generated for serious errors. (A maximum of four warning text words per statement may be saved and then written onto the intermediate storage data set. If more than four warning conditions are encountered, an intermediate text word representing an error is generated and scanning of the statement is terminated.)

The source statement scan for the following READ statement is illustrated in Chart 19.

```
READ (5,10) A,B(1),(C(I),I=1,10),D
```

Chart 19. READ Statement Scan Logic



a(xxxx): Indicates the address of the symbol within parentheses.

adjective code field: A field of an intermediate text entry that contains either an adjective code assigned by the compiler or an actual machine operation code.

allocation table: Used in Phase 7 to determine the amount of additional main storage required by the compiler.

argument list: A list containing the addresses of arguments constructed when an adjective code indicating a call to a subprogram or statement function is detected.

argument list table: Used at object-time to provide the starting address of the argument list for each subprogram or statement function called.

base value table: Used at object-time to obtain base register values.

BLDL table: Provides information necessary for transferring control from one phase to the next for PRFRM compilations.

blocking table: Provides information necessary to deblock compiler input and to block compiler output for PRFRM compilations.

bound variable: An integer variable in a subscript expression that is redefined.

branch list table for SFs and DOs: Used at object-time either by the instructions generated to reference SF expansions or by the instructions generated to control the iteration of DO loops.

branch list table for referenced statement numbers: Used at object-time by the instructions generated to branch to executable statements.

CDL: A portion of the array displacement for subscripted variables.

COMMON text: An internal format used to transmit the information in a COMMON source statement to Phase 12.

communication area: A central gathering area used to communicate information between the various phases of the compiler.

declarative statement: Any one of the following statements: COMMON, DIMENSION, EQUIVALENCE, INTEGER, REAL, DOUBLE PRECISION, EXTERNAL, FORMAT, and SUBROUTINE or FUNCTION.

dictionary: A resident table of the compiler used to store information about symbols used in the source statements. For PRFRM compilations, the dictionary resides in main storage throughout the compilation; for SPACE compilations, the dictionary resides in main storage only through Phase 14.

dictionary index: Consists of pointers to the first entries in the various chains that constitute the dictionary.

end-of-statement indicator: An adjective code that signals the end of a particular statement to a processing phase.

epilog table: Used during Phase 25 when generating the instructions that return the value of variables used as parameters to the calling program.

EQUIVALENCE table: Used by the routines that assign addresses for EQUIVALENCE entries.

EQUIVALENCE text: An internal format used to transmit the information in an EQUIVALENCE source statement to Phase 12.

error: Incorrect usage of the FORTRAN language that may force the end of compilation.

ESD card image: A card image containing an external symbol that is defined or referred to in the source module.

executable statement: A statement that causes the compiler to generate machine instructions.

flush: A compile time I/O request that forces the current output buffer being used for a blocked output data set to be written.

forcing value: A value that indicates an operator's relative position in the hierarchy of operators.

forcing value table: Used during Phase 15 processing to aid in the reordering of intermediate text entries for arithmetic expressions.

hierarchy of operators: Defines the order in which operations must be performed in an arithmetic expression.

interface module: The communications link between the compiler and the operating system.

index mapping table: Used during Phase 20 processing of subscript expressions to maintain a record of all information pertinent to the subscript expression.

interlude: A compiler component that closes and then reopens the various data sets used by the compiler for SPACE compilations. (Interludes do not perform source statement processing.)

intermediate text: An internal representation of the source statements that may eventually be converted to machine language instructions.

internal statement number: A number assigned to each FORTRAN statement by the compiler.

list item: A variable used in a READ or WRITE statement.

load module: The output of the linkage editor; a program in a format suitable for loading into main storage for execution.

location counter: A counter used to assign addresses.

message address table: Used during Phase 30 to aid in the generation of error and warning messages.

message length table: Used during Phase 30 to aid in the generation of error and warning messages.

message text table: Used during Phase 30 to aid in the generation of error and warning messages.

mode/type code field: A field used in the dictionary and intermediate text denoting the mode (real, integer, or double precision) and type (variable, array, function or constant) of a symbol.

object module: The output of a single execution of an assembler or compiler, which constitutes input to the linkage editor.

offset: A calculated indexing factor used to find the correct element in an array for a particular subscript expression.

operations table: A temporary storage area used during Phase 15 processing in the reordering of intermediate text entries for arithmetic expressions.

overflow table: A resident table that contains all dimension, subscript, and statement number information within the source module being compiled.

overflow table index: Consists of pointers to the first entries in the various chains that constitute the overflow table.

p(xxxx): Indicates a pointer to the information (within the parentheses) as represented in the dictionary or the overflow table.

patch table: Used to contain patch records if the patch facility has been enabled and if patch records precede the FORTRAN source module to be compiled.

performance module: Processes compiler I/O requests and end-of-phase requests for PRFRM compilations. The performance module also contains the blocking table and the BLDI table.

phase: Performs compiler initialization or actual source statement processing.

pointer field: The last two bytes of an intermediate text word. It normally contains a relative pointer to a dictionary or overflow table entry.

print buffer module: Contains two I/O buffers for SYSIN and SYSPRINT.

resident table: A table that remains in main storage throughout an entire compilation or throughout a part of a compilation. (The dictionary is resident only up to the end of Phase 14 for SPACE compilations.)

RLD card image: Contains information about an address constant used in the object module.

routine displacement tables: Aid in the location of reserved word processing routines in Phases 10D and 10E.

SEGMAL: A resident table that contains the beginning and ending address of each segment of main storage assigned to the dictionary and overflow table by Phase 7.

SF number: Assigned to each SF definition encountered by Phase 14.

source module: A series of statements in the symbolic language of an assembler or compiler, which constitutes the entire input to a single execution of an assembler or compiler.

subscript table: Temporary storage area used for subscript text encountered during the reordering of intermediate text words by Phase 15.

subscript optimization: The process of replacing the computation of a subscript

expression at each recurrence with a reference to its initial computation (that is, to the register assigned to contain the result of its initial computation).

SYSIN data set: The source module, which is used as input to the compiler.

SYSLIN data set: The object module in card image form (if the LOAD option is specified).

SYSUT1 data set: Used as a work data set by the compiler to contain intermediate text.

SYSUT2 data set: Used as a work data set by the compiler to contain intermediate text.

SYSPRINT data set: The source module listing (if the SOURCE option was specified); a storage map (if the MAP option was specified); and a list of error and warning messages (if any).

SYSPUNCH data set: The object module in card image form (if the DECK option was specified).

SYS1.FORTLIB: A partitioned data set that contains FORTRAN subprograms (including IHCFCOME and IHCFIOSH in the form of load modules.

SYS1.LINKLIB: A partitioned data set that contains executable load modules, which can be reached via the XCTL, ATTACH, LINK, and LOAD functions. The FORTRAN IV (E) compiler resides on the SYS1.LINKLIB.

TXT card image: A card image containing either an instruction of the object module or data used in the object module.

unit assignment table: Used by IHCFIOSH during processing of execution-time I/O requests.

unit blocks: Used by IHCFIOSH during processing of exection time I/O requests.

warning: Incorrect usage of the FORTRAN language that is not serious enough to prevent execution of the object module.

Printed in U.S.A. Y28-6601-1

IBM

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

READER'S COMMENTS

Title: IBM System/360 Operating System
FORTRAN IV (E)
Program Logic Manual

Form: Y28-6601-1

Is the material:	Yes	No
Easy to Read?	___	___
Well organized?	___	___
Complete?	___	___
Well illustrated?	___	___
Accurate?	___	___
Suitable for its intended audience?	___	___

How did you use this publication?
___ As an introduction to the subject ___ For additional knowledge
Other _____

fold

Please check the items that describe your position:
___ Customer personnel ___ Operator ___ Sales Representative
___ IBM personnel ___ Programmer ___ Systems Engineer
___ Manager ___ Customer Engineer ___ Trainee
___ Systems Analyst ___ Instructor Other_____

Please check specific criticism(s), give page number(s),and explain below:
___ Clarification on page(s)
___ Addition on page(s)
___ Deletion on page(s)
___ Error on page(s)

Explanation:

CUT ALONG LINE

fold

Name_____
Company_____
Address_____
City _____
State _____Zip Code _____

FOLD ON TWO LINES,STAPLE AND MAIL
No Postage Necessary if Mailed in U.S.A.

staple

fold fold

FIRST CLASS
PERMIT NO. 81

POUGHKEEPSIE, N.Y.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

POSTAGE WILL BE PAID BY

IBM CORPORATION
P.O. BOX 390
POUGHKEEPSIE, N. Y. 12602

ATTN: PROGRAMMING SYSTEMS PUBLICATIONS
DEPT. D58

fold fold

Printed in U.S.A. Y28-6601-1

IBM®

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

staple